OptiInfra Architecture Overview

Version: 1.0
Date: October 2025
Status: Foundation Document - Write First!

Document Control

Version	Date	Author	Changes
1.0	Oct 2025	OptiInfra Team	Initial architecture document

Document Purpose:
This document serves as the **technical bible** for OptiInfra - a comprehensive architecture specification that must be completed before any coding begins. It provides complete design details for all four AI agents and enables parallel development by multiple teams.

Target Audience:

- Engineering teams building the agents
- Architects reviewing system design
- Product managers understanding capabilities
- QA teams developing test strategies

Table of Contents

---

1. Executive Summary

1.1 Vision Statement

OptiInfra is an **AI-powered infrastructure optimization platform** specifically designed for Large Language Model (LLM) inference workloads. It deploys intelligent agents that continuously monitor, analyze, and optimize infrastructure to reduce costs, improve performance, and maximize resource utilization - all while operating autonomously with minimal human intervention.

1.2 The Problem We Solve

Organizations running LLM inference at scale face critical challenges:

Cost Explosion:

- LLM inference costs 5-10x more than training
- GPU costs are 80%+ of total infrastructure spend
- Manual optimization requires deep expertise and constant attention
- Suboptimal configurations waste 40-60% of infrastructure budget

Performance Bottlenecks:

- SLO breaches due to traffic spikes or resource constraints
- Poor load balancing across instances
- Latency issues impacting user experience
- Inability to respond quickly to changing demand

Resource Inefficiency:

- Memory constraints limiting context window sizes
- KV cache pressure causing OOM errors
- Underutilized GPUs running expensive instances
- Lack of visibility into resource usage patterns

Application Blindness:

- Infrastructure teams don't understand application behavior
- Prompts, models, and usage patterns are opaque
- Optimization opportunities missed at application layer
- No feedback loop between infrastructure and application

1.3 Our Solution: Four Intelligent Agents

OptiInfra deploys **four specialized AI agents** that work together to optimize your LLM infrastructure:

Agent 1: Cost Optimization Agent

Mission: Minimize infrastructure costs without compromising performance

Core Capabilities:

- Automated spot instance migration (40-70% cost savings)
- Right-sizing recommendations based on actual usage
- Reserved instance optimization
- Auto-scaling policy tuning

Impact: 30-50% cost reduction in first 90 days

Agent 2: Performance Optimization Agent

Mission: Maintain and improve SLO compliance through intelligent scaling

Core Capabilities:

- Real-time SLO monitoring and enforcement
- Predictive scaling before breaches occur
- Load balancing optimization
- Latency and throughput tuning

Impact: 95%+ SLO compliance, reduced latency

Agent 3: Resource Optimization Agent

Mission: Maximize GPU memory efficiency through KVOptkit integration

Core Capabilities:

- Memory pressure detection and resolution
- KVOptkit tiered memory management (GPU ↔ CPU ↔ Disk)
- Context window optimization
- OOM prevention

Impact: 2-4x context window increase, 30-40% more throughput

Agent 4: Application Optimization Agent

Mission: Optimize at the application layer with privacy-first approach

Core Capabilities:

- Prompt efficiency analysis (log-based, privacy-preserving)
- Model routing optimization
- RAG pipeline tuning
- Bi-directional optimization with infrastructure agents

Impact: 20-30% efficiency gains at application layer

1.4 Key Differentiators

1. LLM-Native Design

- Purpose-built for LLM inference, not general compute
- Deep integration with vLLM, KVOptkit, and LLM frameworks
- Understands LLM-specific patterns and requirements

2. Four Specialized Agents

- Each agent is an expert in its domain
- Collaborative optimization across cost, performance, resource, and application
- Avoids conflicts through intelligent coordination

3. Truly Autonomous

- Agents make decisions using GPT-4o reasoning
- Continuous learning from outcomes
- Minimal human intervention required (human-in-the-loop optional)

4. Privacy-First Application Optimization

- Tiered privacy model (logs → metrics → deep integration)
- Trust-building approach
- No proprietary data exposure required

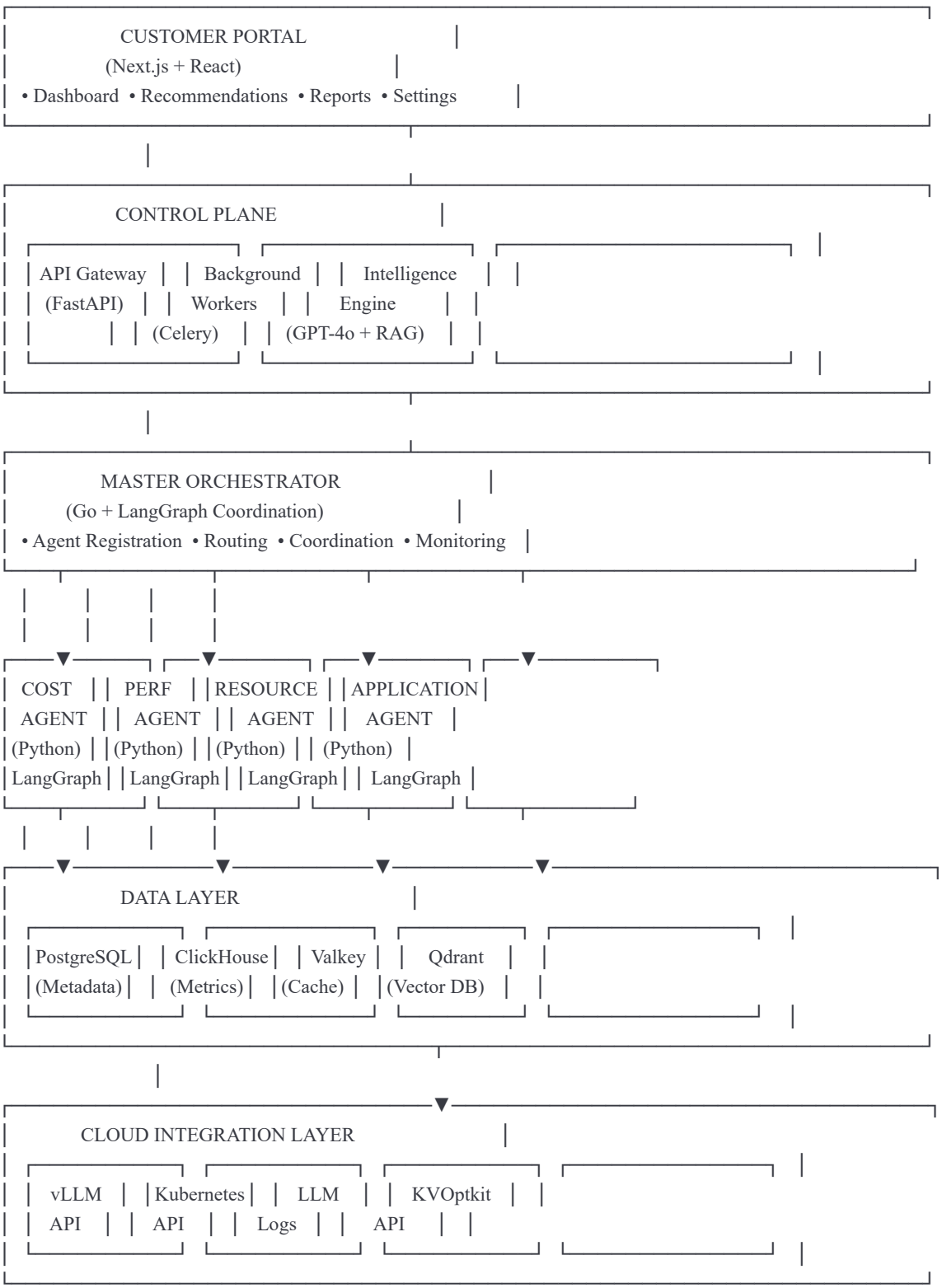
5. Cloud-Agnostic

- Works with any cloud provider (AWS, GCP, Azure, on-prem)
- Standardized APIs for portability
- Mock providers for development and testing

6. Production-Ready Architecture

- Built on battle-tested components (PostgreSQL, ClickHouse, LangGraph)
- Comprehensive testing strategy (400+ tests)
- Enterprise security and compliance

1.5 High-Level Architecture



1.6 Technology Foundation

Languages:

- **Go:** Master Orchestrator (performance, concurrency)
- **Python:** AI Agents (LangGraph, LLM integration)
- **TypeScript/React:** Customer Portal

AI Framework:

- **LangGraph:** Agentic workflow orchestration
- **GPT-4o:** Agent reasoning and decision-making

- **Qdrant:** Vector database for learning and RAG

Data Layer:

- **PostgreSQL:** Transactional data, agent state
- **ClickHouse:** Time-series metrics (billions of data points)
- **Valkey:** Caching and real-time data
- **Qdrant:** Vector embeddings, historical learnings

Infrastructure:

- **Docker/Kubernetes:** Containerization and orchestration
- **FastAPI:** REST APIs
- **Celery:** Background job processing

1.7 Development Approach

Parallel Development Strategy:

- **5 concurrent teams** building simultaneously
- **Month 1:** Shared foundation (orchestrator, databases, mocks)
- **Months 2-4:** Parallel agent development (4 agents + portal)
- **Months 5-6:** Integration, testing, launch preparation
- **Total timeline:** 6 months to production-ready system

Quality Assurance:

- **400-500 total tests** across all agents
- **80%+ coverage** per agent
- **Golden datasets** for LLM testing
- **Integration testing** for multi-agent scenarios

1.8 Success Metrics

Customer Impact:

- 30-50% cost reduction (Cost Agent)
- 95%+ SLO compliance (Performance Agent)
- 2-4x context window increase (Resource Agent)
- 20-30% application efficiency gains (Application Agent)

System Metrics:

- 99.9% agent uptime
- <5 minute response time for critical optimizations
- <1% false positive rate on recommendations
- 90%+ customer acceptance rate for autonomous actions

1.9 Document Roadmap

This architecture document provides:

- **Complete technical specifications** for all four agents (Sections 6-9)
- **System design details** for shared components (Sections 2-5, 10-17)
- **Multi-agent coordination patterns** (Section 19)
- **Data models and APIs** (Section 12)
- **Development roadmap** for parallel building (Section 18)
- **Extensibility framework** for future agents (Section 20)

Read this document if you need to:

- Understand the complete system architecture
- Build any of the four agents
- Integrate with OptiInfra APIs
- Design new features or capabilities
- Review technical decisions

---

2. System Overview

2.1 System Context

OptiInfra operates within a customer's LLM inference infrastructure, serving as an **intelligent automation layer** between infrastructure resources and application workloads.

2.1.1 External Actors

Primary Users:

- **Infrastructure Engineers:** Configure and monitor OptiInfra
- **Platform Operators:** Review and approve recommendations
- **Finance Teams:** Track cost savings and ROI
- **Application Developers:** Receive application optimization insights

External Systems:

- **Cloud Providers:** AWS, GCP, Azure (via standardized APIs)
- **vLLM Instances:** LLM inference engines being optimized
- **Kubernetes Clusters:** Container orchestration platform
- **LLM APIs:** OpenAI/Claude for agent reasoning
- **Logging Systems:** Application logs for analysis
- **Monitoring Systems:** Prometheus, Grafana, etc.

2.1.2 System Boundaries

What OptiInfra Does:

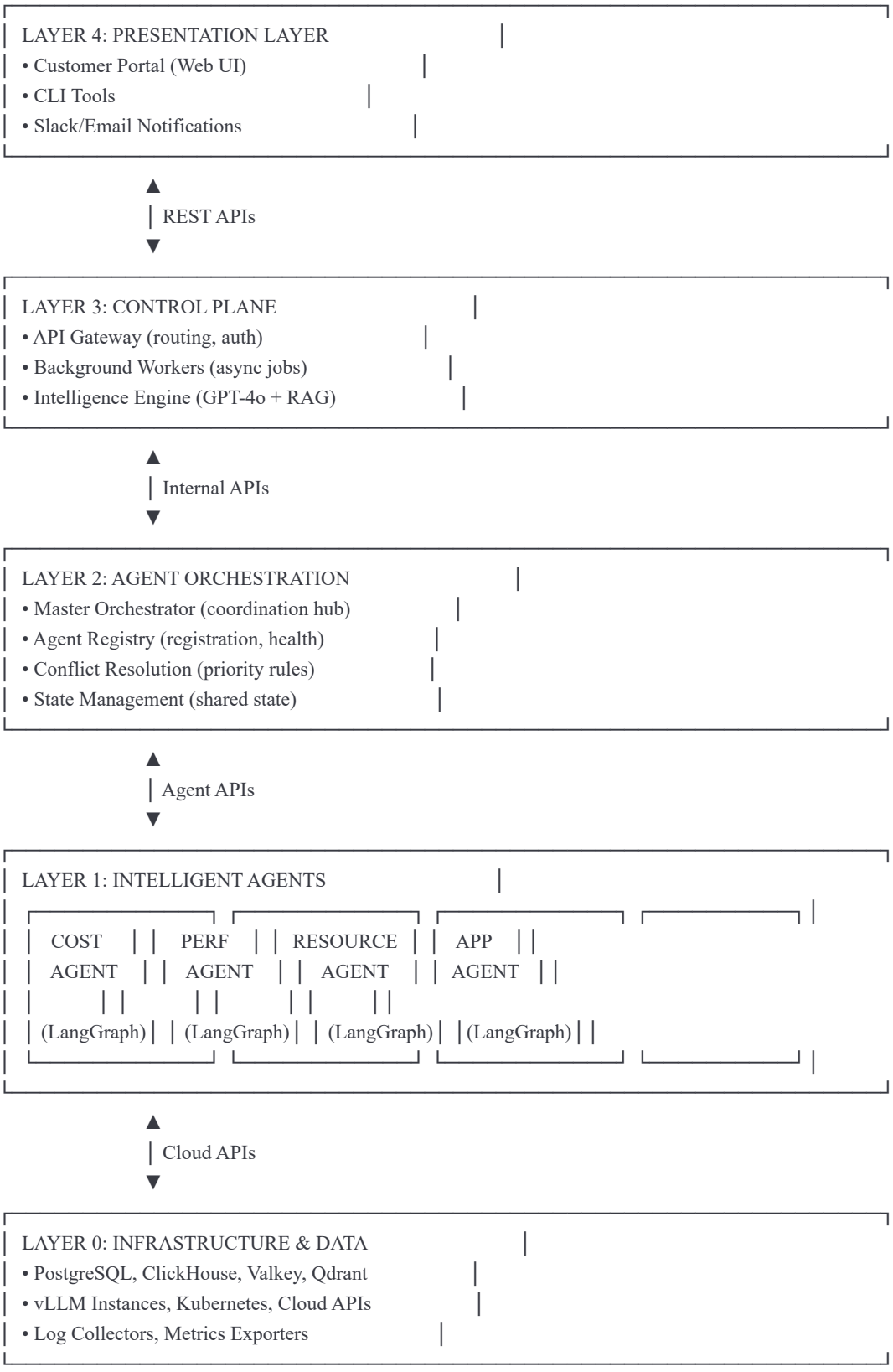
- ✅ Monitors infrastructure and application metrics
- ✅ Analyzes patterns and identifies optimization opportunities
- ✅ Generates and executes optimization recommendations
- ✅ Learns from outcomes and improves over time
- ✅ Coordinates multiple optimization dimensions

What OptiInfra Does NOT Do:

- ❌ Replace existing monitoring tools (integrates with them)
- ❌ Manage Kubernetes directly (uses K8s APIs)
- ❌ Host LLM inference workloads (optimizes existing ones)
- ❌ Store customer application data (privacy-first design)

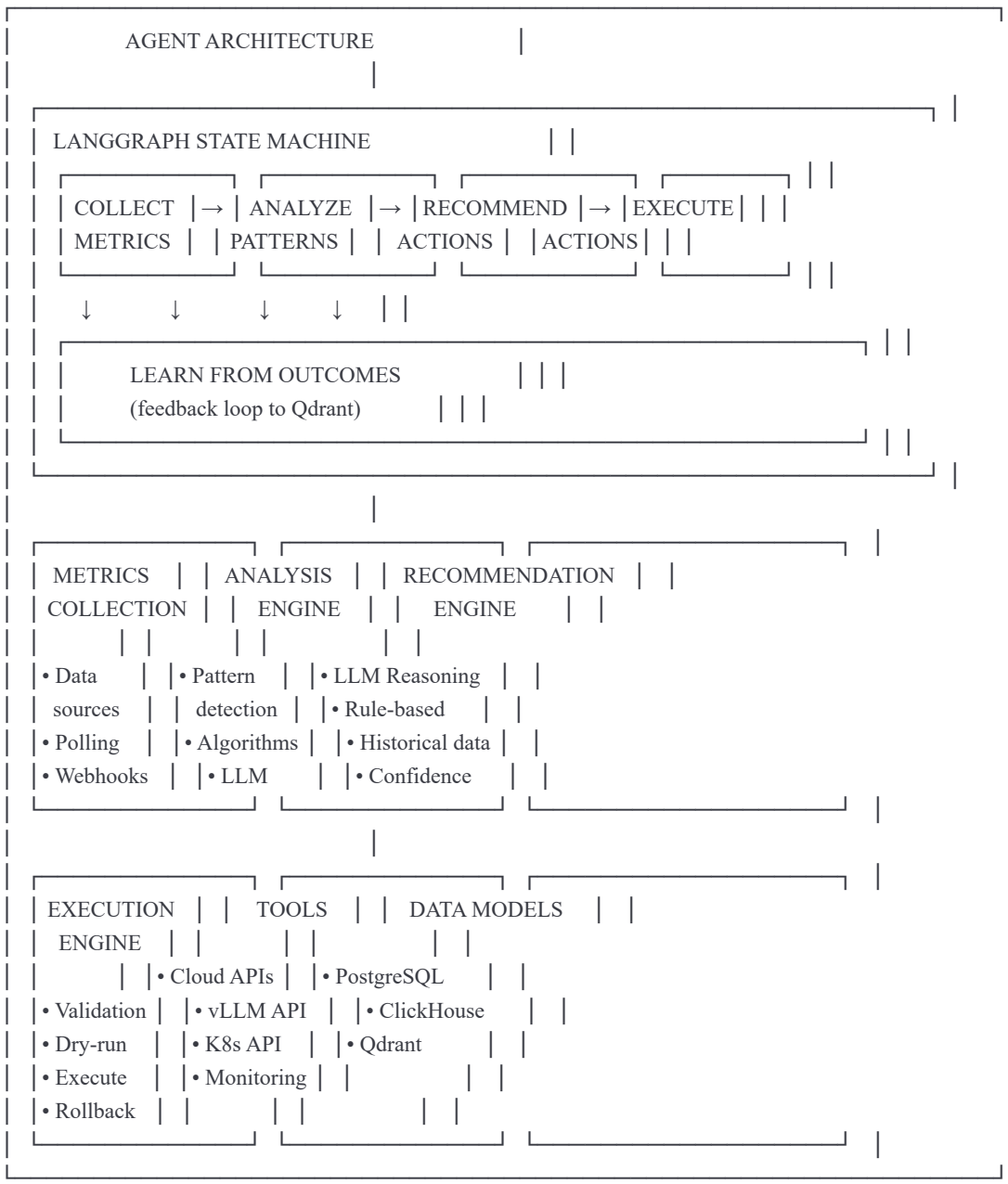
2.2 Conceptual Architecture

2.2.1 Four-Layer Architecture



2.2.2 Agent Architecture Pattern

Each agent follows a consistent architecture pattern:



2.3 Core Components

2.3.1 Master Orchestrator

- **Language:** Go
- **Responsibilities:**
  - Agent registration and health monitoring
  - Request routing to appropriate agents
  - Multi-agent coordination
  - Conflict resolution
  - State management across agents

2.3.2 Four Intelligent Agents

- **Language:** Python + LangGraph
- **Shared Capabilities:**
  - Autonomous decision-making via LLM
  - Continuous learning from outcomes
  - Metrics collection and analysis
  - Recommendation generation
  - Execution with rollback capability

Agent 1: Cost Optimization

- Focus: Minimize infrastructure spend
- Tools: Cloud APIs, spot instance management
- Optimizations: Spot migration, right-sizing, RIs, auto-scaling

Agent 2: Performance Optimization

- Focus: Maintain SLO compliance
- Tools: Kubernetes API, load balancer APIs
- Optimizations: Scaling, load balancing, latency tuning

Agent 3: Resource Optimization

- Focus: Maximize memory efficiency
- Tools: vLLM API, KVOptkit API
- Optimizations: Memory management, KV cache, context windows

Agent 4: Application Optimization

- Focus: Application-layer efficiency
- Tools: Log parsers, metrics APIs
- Optimizations: Prompts, models, RAG pipelines, routing

2.3.3 Control Plane Services

API Gateway:

- REST API endpoints for all operations
- Authentication and authorization
- Rate limiting and throttling
- Request/response logging

Background Workers:

- Async job processing (Celery)
- Scheduled tasks (metrics collection, analysis)
- Report generation
- Alert processing

Intelligence Engine:

- GPT-4o integration for reasoning
- RAG using Qdrant for historical context
- Prompt management and versioning
- LLM response validation

2.3.4 Data Layer

PostgreSQL:

- Transactional data (customers, agents, recommendations)
- Agent state and configuration
- User accounts and permissions
- ACID compliance for critical operations

ClickHouse:

- Time-series metrics (billions of data points)
- High-throughput ingestion
- Fast analytical queries
- Data retention policies

Valkey (Redis fork):

- Real-time caching
- Agent state snapshots
- Rate limiting counters
- Pub/sub for agent communication

Qdrant:

- Vector embeddings of past optimizations
- Semantic search for similar scenarios
- Learning from historical outcomes
- RAG context for LLM reasoning

2.3.5 Customer Portal

- **Technology:** Next.js + React + Tailwind
- **Features:**
  - Real-time dashboard (all 4 agents)
  - Recommendation review and approval
  - Historical reports and analytics
  - Configuration and settings
  - Alert management

2.3.6 Cloud Integration Layer

- Standardized APIs for cloud provider abstraction
- vLLM API clients
- Kubernetes API integration
- Log collection and parsing
- KVOptkit management

2.4 Data Flow

2.4.1 Metrics Collection Flow

1. Agents poll cloud/vLLM/K8s APIs every 60 seconds
2. Raw metrics stored in ClickHouse
3. Aggregated metrics cached in Valkey
4. Agents retrieve metrics for analysis

2.4.2 Optimization Flow

1. Agent detects optimization opportunity
2. Analysis engine processes metrics + LLM reasoning
3. Recommendation generated with confidence score
4. (Optional) Human approval via portal
5. Execution engine validates and executes
6. Outcome tracked and learned from (Qdrant)

2.4.3 Multi-Agent Coordination Flow

1. Multiple agents detect related opportunities
2. Orchestrator receives recommendations
3. Conflict resolution applied (priority rules)
4. Joint optimization computed
5. Coordinated execution across agents
6. Shared learning stored in Qdrant

2.5 Deployment Model

Cloud-Hosted (Primary):

- OptiInfra control plane runs in customer's cloud
- Agents have access to customer's infrastructure APIs

- Data stays in customer's environment
- Customer manages OptiInfra infrastructure

SaaS Model (Future):

- OptiInfra hosted by our team
- Customer provides API credentials
- Data isolated per tenant
- Managed service with SLA

2.6 Operating Modes

2.6.1 Advisory Mode

- Agents generate recommendations only
- No automatic execution
- Human approval required for all actions
- Best for: Initial onboarding, high-risk environments

2.6.2 Semi-Autonomous Mode

- Agents execute low-risk optimizations automatically
- High-impact changes require approval
- Configurable risk thresholds
- Best for: Production environments with oversight

2.6.3 Fully Autonomous Mode

- Agents execute all recommendations automatically
- Human notified of significant changes
- Rollback on negative outcomes
- Best for: Mature deployments with trust established

2.7 Key Design Decisions

Why Four Separate Agents?

- Domain expertise: Each agent masters one optimization dimension
- Parallel development: Teams can build simultaneously
- Extensibility: Easy to add Agent 5, 6, etc.
- Conflict isolation: Agents negotiate rather than interfere

Why LangGraph for Agents?

- Purpose-built for agentic workflows
- State machine visualization
- Built-in checkpointing and error recovery
- Python ecosystem compatibility

Why Go for Orchestrator?

- Performance: Low latency, high concurrency
- Simplicity: Easy to reason about coordination logic
- Reliability: Strong typing, error handling
- Deployment: Single binary, minimal dependencies

Why Four Databases?

- Right tool for each job:
  - PostgreSQL: Transactional integrity
  - ClickHouse: Time-series scale
  - Valkey: Sub-millisecond latency
  - Qdrant: Vector search

---

3. Architectural Principles

3.1 Core Principles

3.1.1 LLM-Native Design

Principle: OptiInfra is purpose-built for LLM inference workloads, not general compute.

Implications:

- Deep understanding of vLLM architecture and metrics
- KV cache optimization is a first-class citizen
- Context window management is critical
- GPU memory patterns are unique to LLMs
- Batch size, sequence length, and token counts matter

Design Decisions:

- Resource Agent dedicated to memory optimization
- Native integration with KVOptkit for tiered memory
- Application Agent understands prompts, models, and RAG patterns
- Metrics focused on LLM-specific dimensions (tokens/sec, KV cache hit rate)

3.1.2 Agent Autonomy with Safety

Principle: Agents operate autonomously but with guardrails and human oversight options.

Implications:

- Agents make decisions using LLM reasoning (GPT-4o)
- Confidence scores guide automatic vs. manual approval
- Dry-run validation before execution
- Automatic rollback on negative outcomes
- Human-in-the-loop for high-risk changes

Design Decisions:

- Three operating modes: Advisory, Semi-Autonomous, Fully Autonomous
- Confidence thresholds configurable per customer
- All actions audited and logged
- Rollback mechanisms for every optimization type

3.1.3 Separation of Concerns

Principle: Each agent owns one optimization dimension; orchestrator manages coordination.

Implications:

- Cost Agent doesn't worry about performance SLOs directly
- Performance Agent doesn't calculate cost implications
- Resource Agent focuses on memory, not pricing
- Application Agent optimizes apps, not infrastructure

Design Decisions:

- Four distinct agents with clear boundaries
- Master Orchestrator for cross-agent coordination
- Conflict resolution at orchestrator level
- Joint optimization strategies when needed

3.1.4 Cloud Agnosticism

Principle: OptiInfra works with any cloud provider or on-premises infrastructure.

Implications:

- No vendor lock-in to AWS, GCP, or Azure
- Portable across environments
- Customer can migrate clouds without OptiInfra refactoring
- Multi-cloud optimization supported

Design Decisions:

- Standardized Cloud Integration Layer (abstraction)
- Provider-specific adapters (AWS, GCP, Azure, Mock)
- Mock cloud provider for development and testing
- Infrastructure-as-Code friendly

3.1.5 Data-Driven Intelligence

Principle: Agents learn from outcomes and improve over time using historical data.

Implications:

- Every optimization outcome tracked
- Similar scenarios retrieved via semantic search
- LLM reasoning augmented with RAG from past learnings
- Confidence improves with more data

Design Decisions:

- Qdrant vector database for semantic storage
- Feedback loop: Action → Outcome → Embedding → Learning
- Historical context included in LLM prompts
- Golden datasets for baseline performance

3.1.6 Privacy First

Principle: Customer data privacy is paramount; trust is earned gradually.

Implications:

- No proprietary data stored unless customer opts in
- Tiered privacy model (logs → metrics → deep integration)
- Application Agent starts with minimal access
- Transparency about data collection

Design Decisions:

- Application Agent Tier 1: Log-based only (no sensitive data)
- Tier 2: Metrics-based (aggregated, anonymized)
- Tier 3: Deep integration (customer grants access)
- All data encrypted at rest and in transit

3.1.7 Test-Driven Quality

Principle: Comprehensive testing at every layer ensures reliability and safety.

Implications:

- Unit tests for every module (80%+ coverage)
- Integration tests for agent workflows
- LLM output validation with golden datasets
- Multi-agent coordination testing
- End-to-end user journey testing

Design Decisions:

- 400-500 total tests across system
- Mock cloud providers for safe testing
- LLM mocking for deterministic tests
- Continuous evaluation of LLM outputs

3.1.8 Extensibility by Design

Principle: New agents can be added without refactoring existing system.

Implications:

- Agent registry pattern for dynamic discovery
- Standardized agent interface contract
- Orchestrator supports N agents, not just 4
- Data models extensible per agent

Design Decisions:

- Agent registration API
- Generic orchestration patterns

- Plugin architecture for agent-specific logic
- Coordination framework scales to many agents

3.2 Operational Principles

3.2.1 Fail-Safe Defaults

- Agents default to advisory mode for new customers
- Confidence thresholds conservative until trust built
- Automatic rollback on unexpected outcomes
- Circuit breakers prevent cascade failures

3.2.2 Gradual Adoption

- Customers start with one agent (usually Cost)
- Additional agents enabled incrementally
- Operating modes progress: Advisory → Semi-Auto → Full Auto
- Trust built through demonstrated value

3.2.3 Observability First

- Every action logged and auditable
- Real-time monitoring of agent health
- Performance metrics for agent execution
- Distributed tracing across agents

3.2.4 Human-Friendly Interfaces

- Natural language explanations for recommendations
- Visual dashboards showing agent activity
- Approval workflows integrated into portal
- Slack/email notifications for significant events

3.3 Technical Principles

3.3.1 Microservices Architecture

- Agents are independent services (can deploy separately)
- Services communicate via REST APIs and message queues
- Each service has its own database schema
- Loose coupling enables parallel development

3.3.2 Event-Driven Updates

- Agents publish events on state changes
- Other agents subscribe to relevant events
- Orchestrator coordinates event flows
- Async processing for non-critical paths

3.3.3 Idempotency

- All agent actions are idempotent (safe to retry)
- Duplicate detection prevents double-execution
- State management ensures consistency
- Recovery from failures is automatic

3.3.4 Scalability

- Agents scale horizontally (multiple instances)
- ClickHouse handles billions of metric data points
- Valkey provides sub-millisecond caching
- Load balancing across agent instances

3.4 Development Principles

3.4.1 Parallel Development

- 5 teams build simultaneously (Agents 1-4 + Portal)
- Shared foundation built first (Month 1)

- Agents developed independently (Months 2-4)
- Integration phase (Months 5-6)

3.4.2 Mock-Driven Development

- Mock cloud providers enable local development
- No cloud accounts needed for development
- Deterministic testing with mocks
- Faster iteration cycles

3.4.3 AI-Assisted Development

- Use Claude/GPT-4 for code generation
- Cursor/GitHub Copilot for autocomplete
- AI generates tests from specifications
- Faster development velocity

3.4.4 Documentation-First

- Architecture documents written before coding
- API specifications precede implementation
- Decision logs capture rationale
- Living documentation updated continuously

---

4. Master Orchestrator

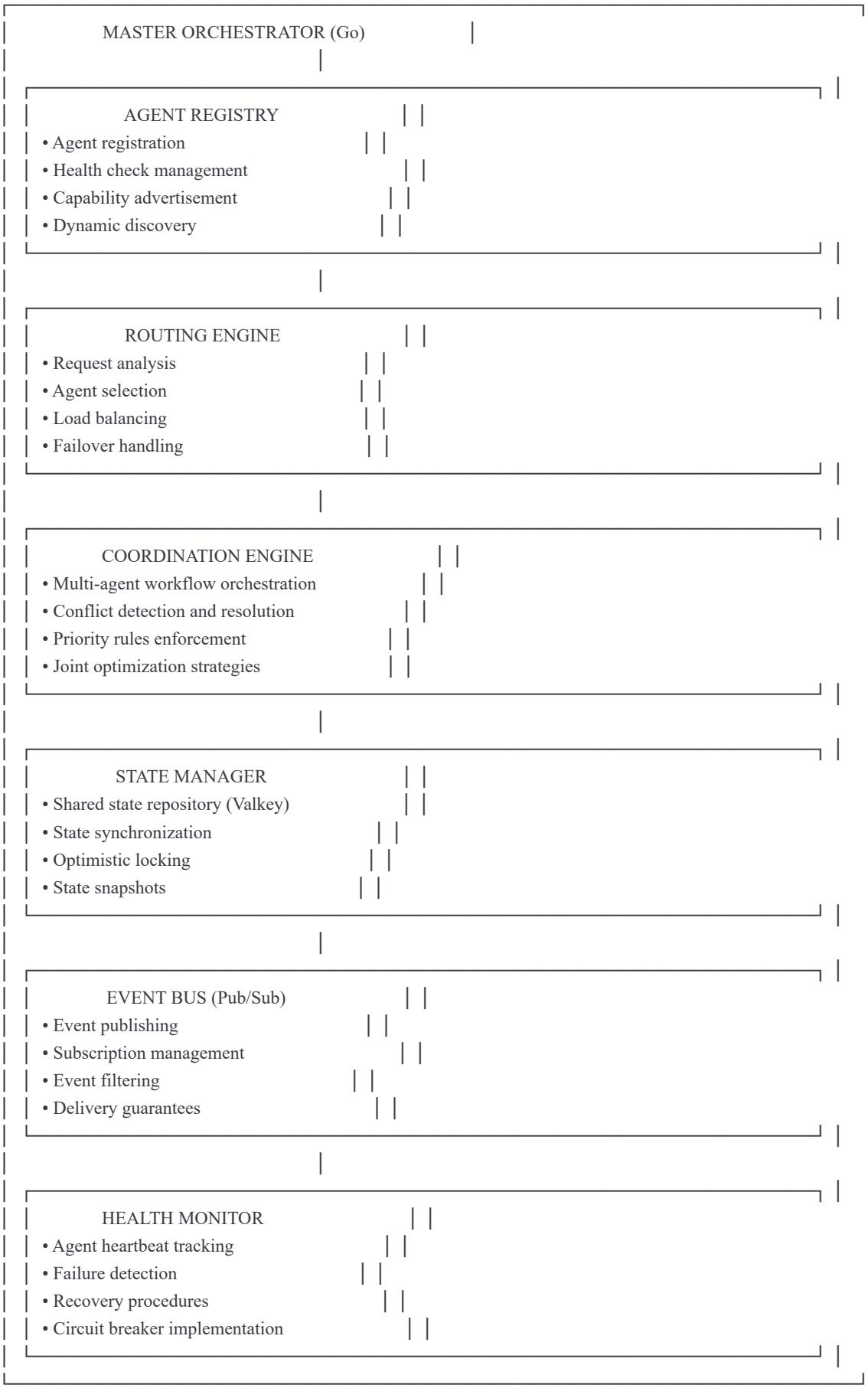
4.1 Overview

The **Master Orchestrator** is the central coordination hub for OptiInfra. Written in Go for performance and reliability, it manages the lifecycle, routing, and coordination of all four intelligent agents.

Core Responsibilities:

1. Agent registration and health monitoring
2. Request routing to appropriate agents
3. Multi-agent coordination and conflict resolution
4. State management across agents
5. Event propagation and pub/sub messaging

4.2 Architecture


MASTER ORCHESTRATOR (Go)
AGENT REGISTRY
• Agent registration
• Health check management
• Capability advertisement
• Dynamic discovery
ROUTING ENGINE
• Request analysis
• Agent selection
• Load balancing
• Failover handling
COORDINATION ENGINE
• Multi-agent workflow orchestration
• Conflict detection and resolution
• Priority rules enforcement
• Joint optimization strategies
STATE MANAGER
• Shared state repository (Valkey)
• State synchronization
• Optimistic locking
• State snapshots
EVENT BUS (Pub/Sub)
• Event publishing
• Subscription management
• Event filtering
• Delivery guarantees
HEALTH MONITOR
• Agent heartbeat tracking
• Failure detection
• Recovery procedures
• Circuit breaker implementation

4.3 Agent Registry

4.3.1 Registration Process

When an agent starts, it registers with the orchestrator:


go

```go
// Agent registration payload
type AgentRegistration struct {
    AgentID      string          // Unique identifier
    AgentType    string          // "cost", "performance", "resource", "application"
    Version      string          // Agent version
    Capabilities []string        // List of capabilities
    Endpoint     string          // HTTP endpoint
    HealthCheck  string          // Health check URL
    Priority     int             // Agent priority (for routing)
    Metadata     map[string]string // Additional metadata
}

// Registration response
type RegistrationResponse struct {
    Success      bool
    OrchestatorID string
    Heartbeat    time.Duration // Expected heartbeat interval
}
```

Registration Flow:

1. Agent sends POST to `/orchestrator/register` with registration payload
2. Orchestrator validates agent type and capabilities
3. Orchestrator stores registration in Valkey (TTL-based)
4. Orchestrator returns acknowledgment with heartbeat interval
5. Agent sends heartbeat every 30 seconds to maintain registration

4.3.2 Capability Advertisement

Each agent advertises its capabilities:

Cost Agent Capabilities:


json

```json
{
  "agent_type": "cost",
  "capabilities": [
    "spot_migration",
    "right_sizing",
    "reserved_instances",
    "auto_scaling",
    "cost_analysis",
    "budget_tracking"
  ]
}
```

Performance Agent Capabilities:


json

```
{
 "agent_type": "performance",
 "capabilities": [
  "slo_monitoring",
  "predictive_scaling",
  "load_balancing",
  "latency_optimization",
  "throughput_tuning"
 ]
}
```

Resource Agent Capabilities:


json

```
{
 "agent_type": "resource",
 "capabilities": [
  "memory_optimization",
  "kvoptkit_deployment",
  "context_window_optimization",
  "oom_prevention",
  "memory_profiling"
 ]
}
```

Application Agent Capabilities:


json

```
{
 "agent_type": "application",
 "capabilities": [
  "prompt_optimization",
  "model_routing",
  "rag_tuning",
  "context_window_sizing",
  "application_profiling"
 ]
}
```

4.3.3 Health Monitoring


go

```
// Health check structure
type HealthStatus struct {
    AgentID   string
    Status    string // "healthy", "degraded", "unhealthy"
    Timestamp time.Time
    Metrics   HealthMetrics
}

type HealthMetrics struct {
    CPU            float64 // CPU usage %
    Memory         float64 // Memory usage %
    ActiveWorkflows int    // Number of active LangGraph workflows
    QueuedTasks    int     // Number of queued tasks
    LastSuccess    time.Time
    ErrorRate      float64 // Errors per minute
}
```

Health Check Frequency:

- Agents send heartbeat every 30 seconds
- Orchestrator marks agent unhealthy after 2 missed heartbeats (60 seconds)
- Unhealthy agents removed from routing pool
- Recovery: Agent registers again when healthy

4.4 Routing Engine

4.4.1 Request Types

The orchestrator handles different request types:

1. Single-Agent Requests

json

```
{
  "request_type": "optimization",
  "target_agent": "cost",
  "operation": "analyze_spot_migration",
  "customer_id": "cust_123",
  "parameters": {
    "instance_ids": ["i-abc123", "i-def456"]
  }
}
```

2. Multi-Agent Requests

json

```json
{
  "request_type": "joint_optimization",
  "target_agents": ["cost", "performance"],
  "operation": "optimize_scaling",
  "customer_id": "cust_123",
  "parameters": {
    "workload_id": "workload_789",
    "constraints": {
      "max_cost_increase": 0.1,
      "min_slo_compliance": 0.95
    }
  }
}
```

3. Query Requests


json

```json
{
  "request_type": "query",
  "target_agent": "cost",
  "operation": "get_savings_report",
  "customer_id": "cust_123",
  "parameters": {
    "start_date": "2025-01-01",
    "end_date": "2025-01-31"
  }
}
```

4.4.2 Routing Algorithm


go

```go
func (r *RoutingEngine) RouteRequest(req *Request) (*Agent, error) {
    // Step 1: Determine required capabilities
    capabilities := r.analyzeRequirements(req)

    // Step 2: Find eligible agents
    eligibleAgents := r.registry.FindByCapabilities(capabilities)

    if len(eligibleAgents) == 0 {
        return nil, errors.New("no agents available for request")
    }

    // Step 3: Filter by health status
    healthyAgents := filterHealthy(eligibleAgents)

    // Step 4: Apply load balancing
    selectedAgent := r.loadBalance(healthyAgents)

    // Step 5: Check circuit breaker
    if r.circuitBreaker.IsOpen(selectedAgent.ID) {
        return r.RouteRequest(req) // Retry with different agent
    }

    return selectedAgent, nil
}

// Load balancing strategies
func (r *RoutingEngine) loadBalance(agents []*Agent) *Agent {
    // Strategy 1: Round-robin (default)
    // Strategy 2: Least loaded (by active workflows)
    // Strategy 3: Random
    // Strategy 4: Priority-based (highest priority first)

    // Using least-loaded strategy
    sort.Slice(agents, func(i, j int) bool {
        return agents[i].Health.ActiveWorkflows < agents[j].Health.ActiveWorkflows
    })

    return agents[0]
}
```

4.4.3 Failover Handling

go

```go
func (r *RoutingEngine) ExecuteWithFailover(req *Request, agent *Agent) (*Response, error) {
    // Try primary agent
    resp, err := r.sendRequest(agent, req)

    if err == nil {
        return resp, nil
    }

    // Primary failed - try failover
    log.Warn("Primary agent failed, attempting failover",
        "agent", agent.ID, "error", err)

    // Mark agent unhealthy temporarily
    r.circuitBreaker.Open(agent.ID, 60*time.Second)

    // Find backup agent
    backupAgent, err := r.RouteRequest(req)
    if err != nil {
        return nil, fmt.Errorf("failover failed: %w", err)
    }

    // Retry with backup
    return r.sendRequest(backupAgent, req)
}
```

4.5 Coordination Engine

4.5.1 Multi-Agent Workflows

The coordination engine orchestrates multi-agent workflows:

Example: Cost + Performance Joint Optimization

go

```go
type JointOptimization struct {
    WorkflowID   string
    Agents       []string // ["cost", "performance"]
    Objective    string   // "minimize_cost_maintain_slo"
    Constraints  map[string]interface{}
    Status       string
    Steps        []OptimizationStep
}

func (c *CoordinationEngine) OptimizeCostAndPerformance(
    customerID string,
    workloadID string,
) (*JointOptimization, error) {

    workflow := &JointOptimization{
        WorkflowID: generateID(),
        Agents:     []string{"cost", "performance"},
        Objective:  "minimize_cost_maintain_slo",
    }

    // Step 1: Cost Agent analyzes current costs
    costAnalysis, err := c.requestAnalysis("cost", customerID, workloadID)
    if err != nil {
        return nil, err
    }

    // Step 2: Performance Agent analyzes current SLOs
    perfAnalysis, err := c.requestAnalysis("performance", customerID, workloadID)
    if err != nil {
        return nil, err
    }

    // Step 3: Detect conflicts
    conflicts := c.detectConflicts(costAnalysis, perfAnalysis)

    if len(conflicts) > 0 {
        // Step 4: Resolve conflicts
        resolution := c.resolveConflicts(conflicts, workflow.Objective)
        workflow.Steps = append(workflow.Steps, resolution)
    }

    // Step 5: Generate joint recommendation
    recommendation := c.generateJointRecommendation(
        costAnalysis,
        perfAnalysis,
        workflow.Constraints,
    )

    // Step 6: Execute coordinated actions
    results := c.executeCoordinated(recommendation)

    workflow.Status = "completed"
    workflow.Steps = append(workflow.Steps, results)
```

```
    return workflow, nil
}
```

4.5.2 Conflict Detection

Conflicts occur when agents have opposing recommendations:

Common Conflict Scenarios:

Scenario	Cost Agent	Performance Agent	Conflict Type
Scale Down vs Scale Up	"Reduce instances to save $500/month"	"Add 2 instances to meet SLO"	Direct opposition
Spot vs On-Demand	"Migrate to spot (70% savings)"	"Use on-demand for reliability"	Risk tolerance
Right-sizing	"Downsize from g4dn.xlarge to medium"	"Keep xlarge for throughput"	Resource allocation

go

```go
type Conflict struct {
    ConflictID   string
    AgentA       string
    AgentB       string
    Type         string // "direct_opposition", "resource_contention", "risk_tradeoff"
    Severity     string // "high", "medium", "low"
    Description  string
    Options      []ConflictOption
}

type ConflictOption struct {
    Description string
    Impact      map[string]float64 // agent -> impact score
    Tradeoffs   string
}

func (c *CoordinationEngine) detectConflicts(
    analysis1 *AgentAnalysis,
    analysis2 *AgentAnalysis,
) []Conflict {
    conflicts := []Conflict{}

    // Example: Cost wants to scale down, Performance wants to scale up
    if analysis1.Recommendation.Action == "scale_down" &&
       analysis2.Recommendation.Action == "scale_up" {

        conflicts = append(conflicts, Conflict{
            ConflictID:  generateID(),
            AgentA:      analysis1.AgentID,
            AgentB:      analysis2.AgentID,
            Type:        "direct_opposition",
            Severity:    "high",
            Description: "Cost Agent wants to scale down, Performance Agent wants to scale up",
        })
    }

    // More conflict detection logic...

    return conflicts
}
```

4.5.3 Conflict Resolution

Resolution Strategies:

1. Priority Rules

go

```go
var DefaultPriorityRules = map[string]int{
    "performance": 1, // Highest priority (SLO compliance critical)
    "resource":    2, // Second (prevent OOM crashes)
    "cost":        3, // Third (cost optimization)
    "application": 4, // Lowest (advisory)
}

func (c *CoordinationEngine) resolveByPriority(conflict Conflict) *Resolution {
    // SLO compliance > Cost savings
    if conflict.AgentA == "performance" || conflict.AgentB == "performance" {
        return &Resolution{
            Winner: "performance",
            Reason: "SLO compliance takes priority over cost optimization",
        }
    }

    // Apply priority rules
    priorityA := DefaultPriorityRules[conflict.AgentA]
    priorityB := DefaultPriorityRules[conflict.AgentB]

    if priorityA < priorityB {
        return &Resolution{Winner: conflict.AgentA}
    }

    return &Resolution{Winner: conflict.AgentB}
}
```

2. Joint Optimization

go

```go
func (c *CoordinationEngine) resolveByOptimization(conflict Conflict) *Resolution {
    // Use LLM to find middle ground
    prompt := fmt.Sprintf(`
        Two agents have conflicting recommendations:

        Cost Agent: %s (Impact: $%0.2f/month saved)
        Performance Agent: %s (Impact: %0.1f%% SLO improvement)

        Find an optimal solution that:
        1. Maintains SLO >= 95%%
        2. Maximizes cost savings within SLO constraint
        3. Provides specific action steps
    `, conflict.Options[0].Description,
       conflict.Options[0].Impact["cost_savings"],
       conflict.Options[1].Description,
       conflict.Options[1].Impact["slo_improvement"])

    llmResponse := c.intelligenceEngine.Query(prompt)

    return &Resolution{
        Strategy: "joint_optimization",
        Action:   llmResponse.Action,
        Reason:   llmResponse.Reasoning,
    }
}
```

3. Constraint Satisfaction

go

```go
func (c *CoordinationEngine) resolveByConstraints(
    conflict Conflict,
    constraints map[string]interface{},
) *Resolution {
    // Example: Customer constraint is "max_cost_increase: 10%"
    maxCostIncrease := constraints["max_cost_increase"].(float64)

    for _, option := range conflict.Options {
        costImpact := option.Impact["cost_change"]

        if costImpact <= maxCostIncrease {
            return &Resolution{
                Strategy: "constraint_satisfaction",
                Action:   option.Description,
                Reason:   "Satisfies customer constraint: max cost increase 10%",
            }
        }
    }

    return nil // No option satisfies constraints
}
```

4.5.4 State Synchronization

go

```go
// Shared state structure
type SharedState struct {
    CustomerID       string
    WorkloadID       string
    CurrentResources map[string]interface{}
    ActiveAgents     []string
    OngoingActions   []Action
    LastUpdate       time.Time
    Version          int // For optimistic locking
}

// State manager operations
func (s *StateManager) GetState(customerID, workloadID string) (*SharedState, error) {
    key := fmt.Sprintf("state:%s:%s", customerID, workloadID)

    data, err := s.valkey.Get(key)
    if err != nil {
        return nil, err
    }

    var state SharedState
    json.Unmarshal(data, &state)

    return &state, nil
}

func (s *StateManager) UpdateState(state *SharedState) error {
    // Optimistic locking
    current, err := s.GetState(state.CustomerID, state.WorkloadID)
    if err != nil {
        return err
    }

    if current.Version != state.Version {
        return errors.New("state version mismatch - retry")
    }

    state.Version++
    state.LastUpdate = time.Now()

    data, _ := json.Marshal(state)
    key := fmt.Sprintf("state:%s:%s", state.CustomerID, state.WorkloadID)

    return s.valkey.Set(key, data, 24*time.Hour)
}
```

4.6 Event Bus

4.6.1 Event Types

go

```
type Event struct {
    EventID     string
    EventType   string // "optimization_started", "optimization_completed", etc.
    SourceAgent string
    Timestamp   time.Time
    Payload     interface{}
    Metadata    map[string]string
}

// Event types
const (
    EventOptimizationStarted   = "optimization_started"
    EventOptimizationCompleted = "optimization_completed"
    EventOptimizationFailed    = "optimization_failed"
    EventMetricsCollected      = "metrics_collected"
    EventRecommendationGenerated = "recommendation_generated"
    EventConflictDetected      = "conflict_detected"
    EventStateChanged          = "state_changed"
)
```

4.6.2 Pub/Sub Implementation

go

```go
type EventBus struct {
    subscribers map[string][]Subscriber
    valkey      *ValKeyClient
    mu          sync.RWMutex
}

type Subscriber struct {
    ID       string
    AgentID  string
    Filter   EventFilter
    Callback func(*Event) error
}

type EventFilter struct {
    EventTypes []string
    SourceAgents []string
    CustomerIDs []string
}

func (eb *EventBus) Publish(event *Event) error {
    // Serialize event
    data, err := json.Marshal(event)
    if err != nil {
        return err
    }

    // Publish to Valkey (Redis) pub/sub
    channel := fmt.Sprintf("events:%s", event.EventType)
    return eb.valkey.Publish(channel, data)
}

func (eb *EventBus) Subscribe(subscriber Subscriber) error {
    eb.mu.Lock()
    defer eb.mu.Unlock()

    for _, eventType := range subscriber.Filter.EventTypes {
        eb.subscribers[eventType] = append(eb.subscribers[eventType], subscriber)
    }

    // Start listening on Valkey channels
    for _, eventType := range subscriber.Filter.EventTypes {
        channel := fmt.Sprintf("events:%s", eventType)
        eb.valkey.Subscribe(channel, subscriber.Callback)
    }

    return nil
}
```

4.6.3 Event Propagation Example

Scenario: Cost Agent completes spot migration

go

```
// Cost Agent publishes event
event := &Event{
    EventID:     generateID(),
    EventType:   EventOptimizationCompleted,
    SourceAgent: "cost_agent_1",
    Timestamp:   time.Now(),
    Payload: map[string]interface{}{
        "optimization_type": "spot_migration",
        "customer_id":       "cust_123",
        "instances_migrated": 5,
        "cost_savings":      "$1200/month",
    },
}

orchestrator.EventBus.Publish(event)

// Performance Agent receives event (subscribed)
performanceAgent.OnEvent(event, func(e *Event) error {
    // Check if spot migration impacts SLOs
    customerID := e.Payload["customer_id"]
    impactAnalysis := performanceAgent.analyzeSLOImpact(customerID)

    if impactAnalysis.SLOAtRisk {
        // Alert orchestrator
        orchestrator.HandleAlert(impactAnalysis)
    }

    return nil
})
```

4.7 API Specification

4.7.1 Core Endpoints

Agent Management:

```
POST   /orchestrator/register          - Register new agent
POST   /orchestrator/heartbeat         - Send heartbeat
GET    /orchestrator/agents            - List all agents
GET    /orchestrator/agents/:id        - Get agent details
DELETE /orchestrator/agents/:id        - Deregister agent
```

Request Routing:

```
POST   /orchestrator/route             - Route request to agent
POST   /orchestrator/optimize          - Trigger optimization
POST   /orchestrator/query             - Query agent for data
```

Coordination:

```
POST   /orchestrator/joint-optimize     - Multi-agent optimization
GET    /orchestrator/conflicts          - List active conflicts
POST   /orchestrator/resolve-conflict   - Resolve specific conflict
GET    /orchestrator/workflows          - List active workflows
```

State Management:

```
GET    /orchestrator/state/:customer/:workload - Get shared state
PUT    /orchestrator/state/:customer/:workload - Update shared state
```

4.7.2 Example API Calls

Register Agent:

bash

```
POST /orchestrator/register
Content-Type: application/json

{
  "agent_id": "cost_agent_1",
  "agent_type": "cost",
  "version": "1.0.0",
  "capabilities": ["spot_migration", "right_sizing", "reserved_instances"],
  "endpoint": "http://cost-agent:8001",
  "health_check": "http://cost-agent:8001/health"
}
```

Joint Optimization:

bash

```
POST /orchestrator/joint-optimize
Content-Type: application/json

{
  "customer_id": "cust_123",
  "workload_id": "workload_789",
  "agents": ["cost", "performance"],
  "objective": "minimize_cost_maintain_slo",
  "constraints": {
    "max_cost_increase": 0.1,
    "min_slo_compliance": 0.95
  }
}
```

4.8 Deployment Configuration

yaml

```yaml
# docker-compose.yml (excerpt)
services:
  orchestrator:
    build: ./orchestrator
    image: optiinfra/orchestrator:latest
    ports:
      - "8000:8000"
    environment:
      - VALKEY_URL=valkey:6379
      - POSTGRES_URL=postgresql://user:pass@postgres:5432/optiinfra
      - LOG_LEVEL=info
      - HEARTBEAT_INTERVAL=30s
      - HEALTH_CHECK_TIMEOUT=60s
    depends_on:
      - valkey
      - postgres
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
```

4.9 Monitoring & Observability

4.9.1 Key Metrics

go

```go
// Prometheus metrics
var (
    registeredAgents = prometheus.NewGauge(prometheus.GaugeOpts{
        Name: "orchestrator_registered_agents",
        Help: "Number of currently registered agents",
    })

    routingLatency = prometheus.NewHistogram(prometheus.HistogramOpts{
        Name: "orchestrator_routing_latency_seconds",
        Help: "Time taken to route requests",
    })

    activeWorkflows = prometheus.NewGauge(prometheus.GaugeOpts{
        Name: "orchestrator_active_workflows",
        Help: "Number of active multi-agent workflows",
    })

    conflictsDetected = prometheus.NewCounter(prometheus.CounterOpts{
        Name: "orchestrator_conflicts_detected",
        Help: "Total number of conflicts detected",
    })
)
```

4.9.2 Logging

go

```go
// Structured logging
log.Info("Agent registered successfully",
    "agent_id", registration.AgentID,
    "agent_type", registration.AgentType,
    "capabilities", registration.Capabilities)

log.Warn("Agent health check failed",
    "agent_id", agentID,
    "missed_heartbeats", count,
    "last_seen", lastSeen)

log.Error("Conflict resolution failed",
    "conflict_id", conflictID,
    "agents", conflict.Agents,
    "error", err)
```

---

5. Agentic AI Framework - LangGraph

5.1 Why LangGraph?

LangGraph is the foundation for all four OptiInfra agents. It provides:

1. State Machine Orchestration

- Agents model workflows as directed graphs
- Clear state transitions and decision points

- Built-in checkpointing for fault tolerance
- Visual workflow representation

2. LLM Integration

- First-class support for LLM calls
- Prompt management and versioning
- Streaming responses
- Error handling for LLM failures

3. Human-in-the-Loop

- Approval gates for high-risk actions
- Interactive workflows
- Feedback collection

4. Python Ecosystem

- Access to ML/AI libraries (scikit-learn, pandas, numpy)
- vLLM, KVOptkit, cloud SDK integration
- Rich testing frameworks

5.2 LangGraph Core Concepts

5.2.1 State

Every agent maintains a typed state object:


python

```python
from typing import TypedDict, List, Optional
from langgraph.graph import StateGraph

class AgentState(TypedDict):
    """Base state for all agents"""
    customer_id: str
    workload_id: str
    workflow_id: str
    current_step: str

    # Data collected
    metrics: dict
    analysis_results: dict

    # Recommendations
    recommendations: List[dict]
    selected_recommendation: Optional[dict]

    # Execution
    execution_status: str
    execution_results: dict

    # LLM interaction
    llm_prompts: List[str]
    llm_responses: List[str]

    # Error handling
    errors: List[str]
    retry_count: int
```

5.2.2 Nodes

Nodes are functions that process state:

python

```python
def collect_metrics(state: AgentState) -> AgentState:
    """Node: Collect metrics from cloud/vLLM APIs"""
    customer_id = state["customer_id"]
    workload_id = state["workload_id"]

    # Collect metrics
    metrics = metrics_collector.collect(customer_id, workload_id)

    # Update state
    state["metrics"] = metrics
    state["current_step"] = "analyze"

    return state

def analyze_patterns(state: AgentState) -> AgentState:
    """Node: Analyze metrics using LLM + algorithms"""
    metrics = state["metrics"]

    # Run analysis engine
    analysis = analysis_engine.analyze(metrics)

    # LLM reasoning
    llm_prompt = f"Analyze these metrics: {metrics}..."
    llm_response = llm.invoke(llm_prompt)

    state["analysis_results"] = analysis
    state["llm_responses"].append(llm_response)
    state["current_step"] = "recommend"

    return state
```

5.2.3 Edges & Conditional Routing

python

```python
def should_execute_automatically(state: AgentState) -> str:
    """Conditional edge: Execute automatically or wait for approval?"""
    recommendation = state["selected_recommendation"]

    # Check confidence score
    if recommendation["confidence"] >= 0.9:
        return "execute"

    # Check risk level
    if recommendation["risk_level"] == "low":
        return "execute"

    # Default: wait for human approval
    return "wait_approval"
```

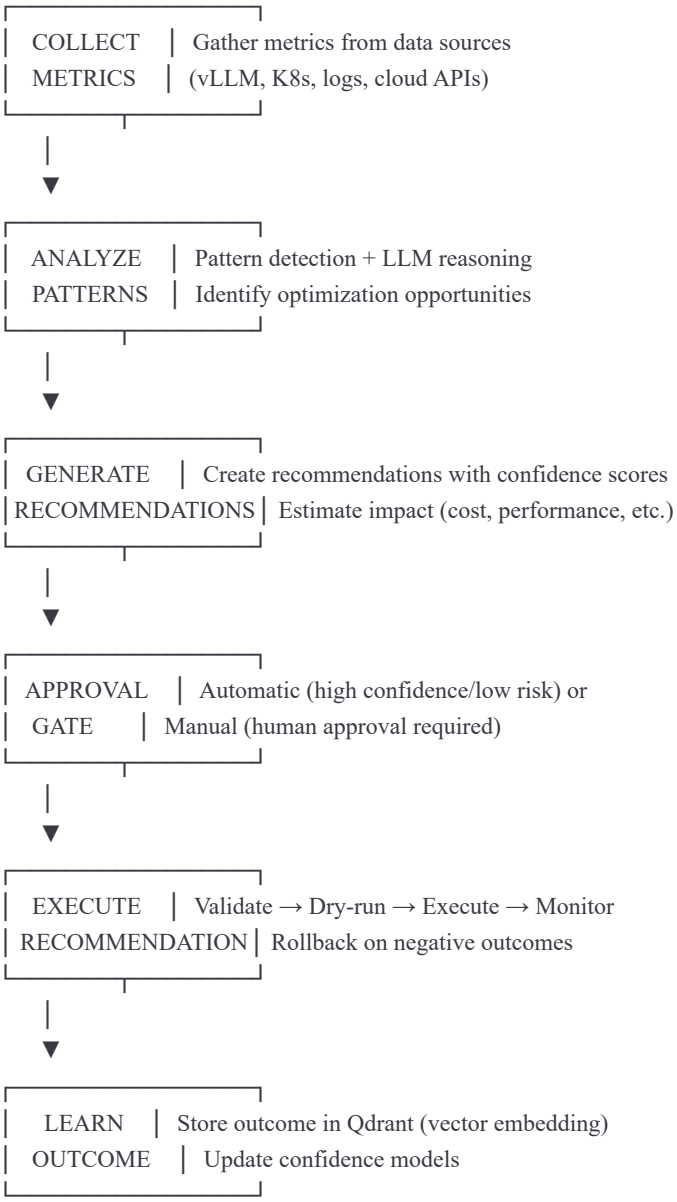
5.2.4 Graph Construction

python

```python
from langgraph.graph import StateGraph, END

# Create graph
workflow = StateGraph(AgentState)

# Add nodes
workflow.add_node("collect_metrics", collect_metrics)
workflow.add_node("analyze_patterns", analyze_patterns)
workflow.add_node("generate_recommendations", generate_recommendations)
workflow.add_node("wait_approval", wait_for_approval)
workflow.add_node("execute_recommendation", execute_recommendation)
workflow.add_node("learn_outcome", learn_from_outcome)

# Add edges
workflow.add_edge("collect_metrics", "analyze_patterns")
workflow.add_edge("analyze_patterns", "generate_recommendations")

# Conditional routing
workflow.add_conditional_edges(
    "generate_recommendations",
    should_execute_automatically,
    {
        "execute": "execute_recommendation",
        "wait_approval": "wait_approval",
    }
)

workflow.add_edge("wait_approval", "execute_recommendation")
workflow.add_edge("execute_recommendation", "learn_outcome")
workflow.add_edge("learn_outcome", END)

# Set entry point
workflow.set_entry_point("collect_metrics")

# Compile graph
app = workflow.compile()
```

5.3 Agent Workflow Pattern

All four agents follow this standard workflow pattern:



5.4 LLM Integration Patterns

5.4.1 Prompt Templates

python

```python
from langchain.prompts import ChatPromptTemplate

# Cost optimization prompt
COST_OPTIMIZATION_PROMPT = ChatPromptTemplate.from_messages([
    ("system", """You are an expert in cloud cost optimization for LLM inference workloads.

    Your task: Analyze the provided metrics and recommend cost optimizations.

    Consider:
    1. Spot instance migration (40-70% savings, some risk)
    2. Right-sizing (reduce waste, maintain performance)
    3. Reserved instances (long-term commitment, guaranteed savings)
    4. Auto-scaling (match capacity to demand)

    Provide:
    - Specific recommendation with action steps
    - Estimated savings ($/month)
    - Risk assessment (low/medium/high)
    - Confidence score (0.0-1.0)
    - Reasoning for your recommendation
    """),

    ("human", """Current Infrastructure:
    - Instances: {instance_count} x {instance_type}
    - Monthly cost: ${monthly_cost}
    - Utilization: {utilization}%
    - Spot-eligible: {spot_eligible}

    Historical Data:
    {historical_context}

    What cost optimization do you recommend?
    """)
])
```

5.4.2 LLM Invocation with Retry

python

```python
from openai import OpenAI
from tenacity import retry, stop_after_attempt, wait_exponential

client = OpenAI(api_key=settings.OPENAI_API_KEY)

@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=1, min=2, max=10),
)
def invoke_llm(prompt: str, model: str = "gpt-4o") -> dict:
    """Invoke LLM with retry logic"""
    try:
        response = client.chat.completions.create(
            model=model,
            messages=[{"role": "user", "content": prompt}],
            temperature=0.7,
            max_tokens=2000,
            response_format={"type": "json_object"},  # Force JSON output
        )

        content = response.choices[0].message.content
        parsed = json.loads(content)

        return {
            "success": True,
            "response": parsed,
            "tokens_used": response.usage.total_tokens,
            "model": model,
        }

    except Exception as e:
        return {
            "success": False,
            "error": str(e),
        }
```

5.4.3 Response Validation

python

```python
from pydantic import BaseModel, Field, validator

class CostRecommendation(BaseModel):
    """Structured LLM response for cost optimization"""
    optimization_type: str = Field(..., description="Type of optimization")
    action_steps: List[str] = Field(..., description="Specific steps to execute")
    estimated_savings: float = Field(..., ge=0, description="Monthly savings in USD")
    risk_level: str = Field(..., description="Risk level: low, medium, high")
    confidence: float = Field(..., ge=0.0, le=1.0, description="Confidence score")
    reasoning: str = Field(..., description="Explanation for recommendation")

    @validator("optimization_type")
    def validate_optimization_type(cls, v):
        allowed = ["spot_migration", "right_sizing", "reserved_instances", "auto_scaling"]
        if v not in allowed:
            raise ValueError(f"Invalid optimization type: {v}")
        return v

def validate_llm_response(response: dict) -> CostRecommendation:
    """Validate and parse LLM response"""
    try:
        return CostRecommendation(**response)
    except Exception as e:
        raise ValueError(f"LLM response validation failed: {e}")
```

5.5 Checkpointing & Recovery

LangGraph provides built-in checkpointing for fault tolerance:

python

```python
from langgraph.checkpoint import MemorySaver, PostgresSaver

# In-memory checkpointing (development)
memory_saver = MemorySaver()

# PostgreSQL checkpointing (production)
postgres_saver = PostgresSaver(connection_string=settings.POSTGRES_URL)

# Compile graph with checkpointing
app = workflow.compile(checkpointer=postgres_saver)

# Execute with checkpointing
config = {"configurable": {"thread_id": workflow_id}}
result = app.invoke(initial_state, config=config)

# Resume from checkpoint on failure
if workflow_failed:
    # Get checkpoint
    checkpoint = postgres_saver.get(thread_id=workflow_id)

    # Resume from last successful state
    result = app.invoke(checkpoint.state, config=config)
```

5.6 Human-in-the-Loop

python

```python
from langgraph.graph import interrupt

def wait_for_approval(state: AgentState) -> AgentState:
    """Node: Wait for human approval before execution"""
    recommendation = state["selected_recommendation"]

    # Store approval request
    approval_id = store_approval_request(
        customer_id=state["customer_id"],
        recommendation=recommendation,
        expires_in=timedelta(hours=24),
    )

    state["approval_id"] = approval_id
    state["current_step"] = "waiting_approval"

    # Interrupt workflow - wait for external signal
    interrupt("waiting_for_approval")

    return state

# Resume workflow after approval
def handle_approval(workflow_id: str, approved: bool):
    """Resume workflow after human decision"""
    config = {"configurable": {"thread_id": workflow_id}}

    # Update state with approval decision
    state = get_workflow_state(workflow_id)
    state["approval_granted"] = approved

    if approved:
        # Resume workflow
        app.invoke(state, config=config)
    else:
        # Cancel workflow
        cancel_workflow(workflow_id)
```

5.7 Testing LangGraph Agents

5.7.1 Unit Testing Nodes

python

```python
import pytest

def test_collect_metrics_node():
    """Test metrics collection node"""
    # Arrange
    state = {
        "customer_id": "cust_123",
        "workload_id": "workload_789",
        "metrics": {},
    }

    # Mock metrics collector
    with patch("agent.metrics_collector.collect") as mock_collect:
        mock_collect.return_value = {"cpu": 0.75, "memory": 0.60}

        # Act
        result = collect_metrics(state)

        # Assert
        assert result["metrics"]["cpu"] == 0.75
        assert result["metrics"]["memory"] == 0.60
        assert result["current_step"] == "analyze"
```

5.7.2 Integration Testing Workflows

python

```python
def test_cost_optimization_workflow():
    """Test complete cost optimization workflow"""
    # Arrange
    initial_state = {
        "customer_id": "cust_123",
        "workload_id": "workload_789",
        "workflow_id": "workflow_001",
        "current_step": "collect_metrics",
        # ... other fields
    }

    # Mock external services
    with patch("agent.cloud_api") as mock_cloud, \
         patch("agent.llm_client") as mock_llm:

        mock_cloud.get_instances.return_value = [...]
        mock_llm.invoke.return_value = {...}

        # Act
        config = {"configurable": {"thread_id": "workflow_001"}}
        result = app.invoke(initial_state, config=config)

        # Assert
        assert result["current_step"] == "completed"
        assert len(result["recommendations"]) > 0
        assert result["execution_status"] == "success"
```

5.7.3 Mocking LLM Responses

python

```python
@pytest.fixture
def mock_llm_response():
    """Fixture for mocked LLM responses"""
    return {
        "optimization_type": "spot_migration",
        "action_steps": [
            "Identify spot-eligible instances",
            "Create spot requests",
            "Monitor spot interruptions",
        ],
        "estimated_savings": 1200.0,
        "risk_level": "medium",
        "confidence": 0.85,
        "reasoning": "Historical data shows 95% spot availability for this region/instance type."
    }

def test_analyze_with_mocked_llm(mock_llm_response):
    """Test analysis node with mocked LLM"""
    with patch("agent.llm.invoke") as mock_invoke:
        mock_invoke.return_value = {"success": True, "response": mock_llm_response}

        state = {...}
        result = analyze_patterns(state)

        assert result["analysis_results"]["optimization_type"] == "spot_migration"
```

5.8 Agent Deployment

5.8.1 Docker Configuration

dockerfile

```
# Dockerfile for Python agents
FROM python:3.11-slim

WORKDIR /app

# Install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy agent code
COPY . .

# Expose port
EXPOSE 8001

# Health check
HEALTHCHECK --interval=30s --timeout=10s --retries=3 \
  CMD python -c "import requests; requests.get('http://localhost:8001/health')"

# Run agent
CMD ["python", "main.py"]
```

5.8.2 Agent Service Template

python

```python
from fastapi import FastAPI, BackgroundTasks
from langgraph.graph import StateGraph

app = FastAPI(title="Cost Optimization Agent")

# Initialize LangGraph workflow
workflow = build_cost_agent_workflow()
compiled_app = workflow.compile(checkpointer=postgres_saver)

@app.post("/optimize")
async def trigger_optimization(
    request: OptimizationRequest,
    background_tasks: BackgroundTasks,
):
    """Trigger cost optimization workflow"""
    workflow_id = generate_id()

    initial_state = {
        "customer_id": request.customer_id,
        "workload_id": request.workload_id,
        "workflow_id": workflow_id,
        # ... initialize state
    }

    # Run workflow in background
    background_tasks.add_task(
        run_workflow,
        compiled_app,
        initial_state,
        workflow_id,
    )

    return {"workflow_id": workflow_id, "status": "started"}

@app.get("/workflow/{workflow_id}")
async def get_workflow_status(workflow_id: str):
    """Get workflow status"""
    state = get_workflow_state(workflow_id)
    return {
        "workflow_id": workflow_id,
        "current_step": state["current_step"],
        "status": state["execution_status"],
    }

@app.get("/health")
async def health_check():
    """Health check endpoint"""
    return {"status": "healthy", "agent": "cost"}

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8001)
```

6. Cost Agent - DETAILED

6.1 Agent Overview

The **Cost Optimization Agent** is the first and most impactful agent in OptiInfra. Its mission is to minimize infrastructure costs for LLM inference workloads while maintaining performance requirements.

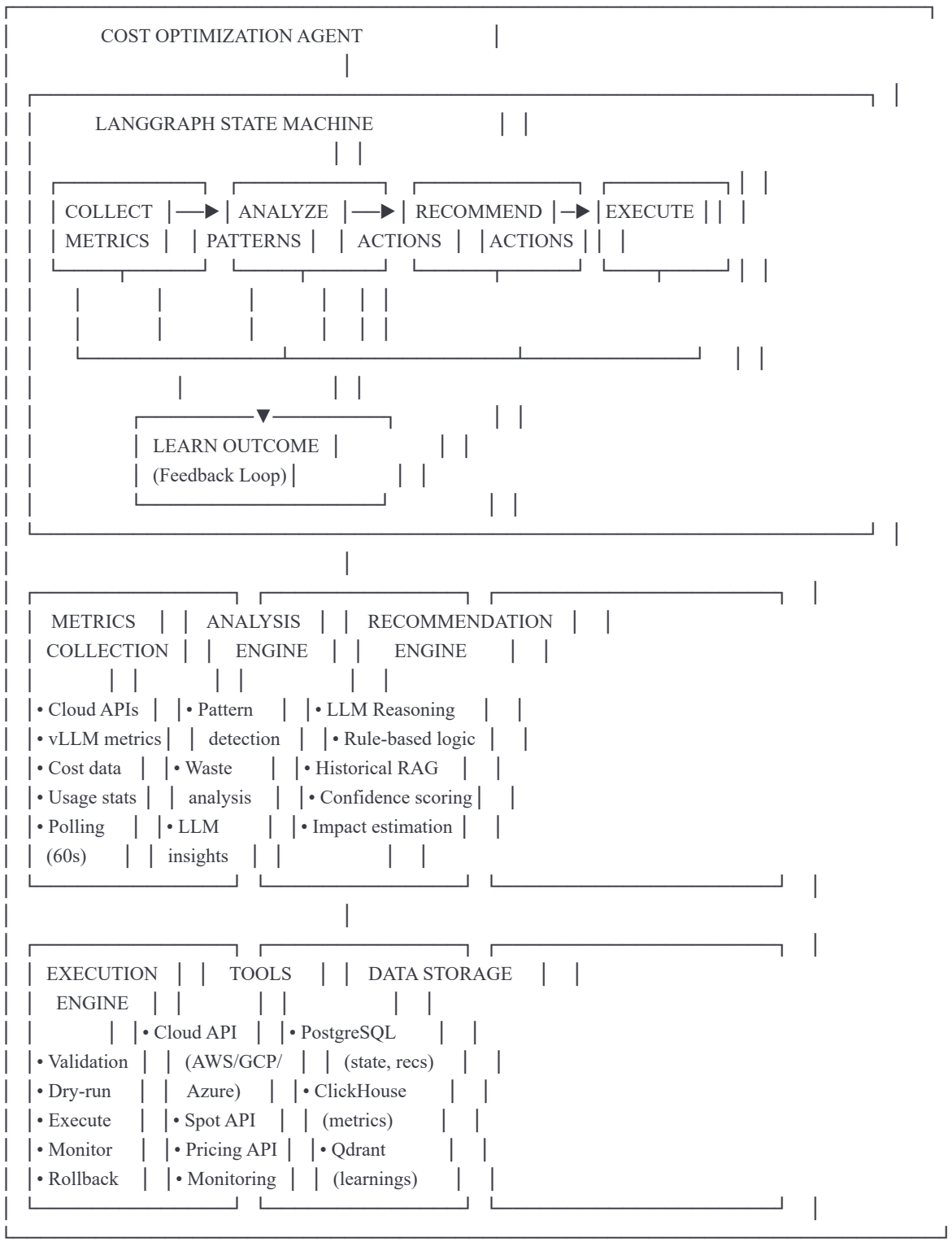
Primary Objectives:

1. Reduce monthly infrastructure spend by 30-50%
2. Identify and execute cost optimization opportunities automatically
3. Learn from outcomes to improve recommendations over time
4. Coordinate with Performance and Resource agents to avoid conflicts

Target Savings:

- **Spot Migration:** 40-70% savings on compute costs
- **Right-sizing:** 20-30% savings by eliminating waste
- **Reserved Instances:** 30-50% savings via long-term commitments
- **Auto-scaling:** 15-25% savings by matching capacity to demand

6.2 Complete Architecture



6.3 Four Optimization Strategies

6.3.1 Strategy 1: Spot Instance Migration

Concept: Migrate on-demand instances to spot instances for 40-70% cost savings.

Algorithm:

python

```python
def analyze_spot_migration_opportunity(instances: List[Instance]) -> SpotAnalysis:
    """
    Analyze spot migration potential for given instances

    Steps:
    1. Check spot eligibility (stateless, fault-tolerant workloads)
    2. Calculate historical spot availability (last 90 days)
    3. Estimate cost savings
    4. Assess risk based on interruption rates
    5. Generate recommendation with confidence score
    """

    eligible_instances = []

    for instance in instances:
        # Check eligibility
        if not is_spot_eligible(instance):
            continue

        # Get historical spot availability
        availability = get_spot_availability(
            instance.type,
            instance.region,
            days=90
        )

        # Calculate savings
        on_demand_cost = instance.hourly_rate * 730  # per month
        spot_cost = instance.spot_rate * 730
        monthly_savings = on_demand_cost - spot_cost
        savings_percent = (monthly_savings / on_demand_cost) * 100

        # Assess risk
        interruption_rate = 1 - availability
        risk_level = calculate_risk_level(interruption_rate)

        # Confidence score
        confidence = calculate_confidence(
            availability=availability,
            historical_data_points=len(availability.data),
            workload_type=instance.workload_type,
        )

        eligible_instances.append({
            "instance_id": instance.id,
            "monthly_savings": monthly_savings,
            "savings_percent": savings_percent,
            "risk_level": risk_level,
            "confidence": confidence,
            "spot_availability": availability.mean,
        })

    return SpotAnalysis(
        eligible_count=len(eligible_instances),
        total_monthly_savings=sum(i["monthly_savings"] for i in eligible_instances),
```

```
        instances=eligible_instances,
    )

def is_spot_eligible(instance: Instance) -> bool:
    """Determine if instance is eligible for spot"""
    # Criteria for spot eligibility:
    # 1. Workload is stateless (no persistent state on instance)
    # 2. Workload is fault-tolerant (can handle interruptions)
    # 3. Not running critical real-time services
    # 4. Has auto-scaling group for automatic replacement

    if instance.tags.get("critical") == "true":
        return False

    if not instance.auto_scaling_enabled:
        return False

    # LLM inference is typically spot-eligible
    if instance.tags.get("workload") == "llm_inference":
        return True

    return False

def calculate_risk_level(interruption_rate: float) -> str:
    """Map interruption rate to risk level"""
    if interruption_rate < 0.05:  # <5% interruption
        return "low"
    elif interruption_rate < 0.15:  # 5-15%
        return "medium"
    else:
        return "high"
```

LLM Prompt for Spot Migration:

python

```
SPOT_MIGRATION_PROMPT = """You are a cloud cost optimization expert specializing in spot instance strategies.

Current Situation:
- Customer: {customer_name}
- Region: {region}
- Instance Type: {instance_type}
- Current Cost: ${on_demand_cost}/month (on-demand)
- Potential Spot Cost: ${spot_cost}/month
- Potential Savings: ${savings}/month ({savings_percent}%)

Historical Data (Last 90 Days):
- Spot Availability: {availability}%
- Interruption Rate: {interruption_rate}%
- Longest Continuous Period: {longest_period} hours

Similar Scenarios (from past learnings):
{historical_context}

Your Task:
1. Assess the risk of spot migration for this workload
2. Recommend whether to migrate to spot instances
3. If yes, suggest specific migration steps and safeguards
4. Estimate confidence in your recommendation (0.0-1.0)

Provide response in JSON format:
{
  "recommendation": "migrate" | "dont_migrate",
  "reasoning": "...",
  "confidence": 0.0-1.0,
  "risk_assessment": "low" | "medium" | "high",
  "migration_steps": ["step 1", "step 2", ...],
  "safeguards": ["safeguard 1", "safeguard 2", ...],
  "estimated_savings": dollars per month
}
"""
```

Execution Steps:

python

```python
def execute_spot_migration(recommendation: SpotRecommendation) -> ExecutionResult:
    """Execute spot instance migration"""

    try:
        # Step 1: Validate recommendation
        validate_recommendation(recommendation)

        # Step 2: Create spot request
        spot_request = create_spot_request(
            instance_type=recommendation.instance_type,
            max_price=recommendation.max_spot_price,
            availability_zone=recommendation.az,
        )

        # Step 3: Wait for spot fulfillment
        spot_instance = wait_for_spot_fulfillment(spot_request, timeout=300)

        # Step 4: Health check new spot instance
        if not health_check(spot_instance):
            rollback_spot_migration(spot_instance, recommendation.original_instance)
            return ExecutionResult(success=False, reason="Health check failed")

        # Step 5: Drain traffic from on-demand to spot
        drain_traffic(
            from_instance=recommendation.original_instance,
            to_instance=spot_instance,
            duration=60,  # Gradual transition
        )

        # Step 6: Terminate on-demand instance
        terminate_instance(recommendation.original_instance)

        # Step 7: Monitor for 1 hour
        monitor_result = monitor_spot_instance(spot_instance, duration=3600)

        if not monitor_result.stable:
            # Rollback if unstable
            rollback_spot_migration(spot_instance, recommendation.original_instance)
            return ExecutionResult(success=False, reason="Instability detected")

        # Step 8: Record success
        actual_savings = calculate_actual_savings(
            old_cost=recommendation.on_demand_cost,
            new_cost=get_instance_cost(spot_instance),
        )

        return ExecutionResult(
            success=True,
            spot_instance_id=spot_instance.id,
            actual_savings=actual_savings,
            execution_time=time.time() - start_time,
        )

    except Exception as e:
        # Rollback on any error
```

```
        rollback_spot_migration(spot_instance, recommendation.original_instance)
        return ExecutionResult(success=False, error=str(e))
```

6.3.2 Strategy 2: Right-Sizing

Concept: Reduce instance sizes to match actual resource usage, eliminating waste.

Algorithm:

python

```python
def analyze_right_sizing_opportunity(instance: Instance) -> RightSizingAnalysis:
    """
    Analyze right-sizing potential

    Steps:
    1. Collect utilization metrics (30 days)
    2. Calculate p95 utilization for CPU, memory, GPU
    3. Find smaller instance type that fits p95
    4. Estimate cost savings
    5. Assess performance risk
    """

    # Get 30 days of metrics
    metrics = get_instance_metrics(
        instance_id=instance.id,
        days=30,
        metrics=["cpu", "memory", "gpu_memory", "gpu_utilization"],
    )

    # Calculate p95 utilization
    p95_cpu = np.percentile(metrics["cpu"], 95)
    p95_memory = np.percentile(metrics["memory"], 95)
    p95_gpu = np.percentile(metrics["gpu_utilization"], 95)
    p95_gpu_memory = np.percentile(metrics["gpu_memory"], 95)

    # Current instance specs
    current_specs = get_instance_specs(instance.type)

    # Calculate waste
    cpu_waste = (current_specs.cpu - p95_cpu) / current_specs.cpu
    memory_waste = (current_specs.memory - p95_memory) / current_specs.memory

    # Find optimal instance size
    optimal_instance = find_optimal_instance_type(
        required_cpu=p95_cpu * 1.2,  # 20% headroom
        required_memory=p95_memory * 1.2,
        required_gpu=instance.gpu_count,
        required_gpu_memory=p95_gpu_memory * 1.2,
    )

    # Calculate savings
    current_cost = instance.hourly_rate * 730
    optimal_cost = optimal_instance.hourly_rate * 730
    monthly_savings = current_cost - optimal_cost

    # Assess risk
    risk = assess_right_sizing_risk(
        current_p95={"cpu": p95_cpu, "memory": p95_memory},
        new_capacity=optimal_instance.specs,
        headroom=0.2,
    )

    return RightSizingAnalysis(
        current_instance=instance.type,
        optimal_instance=optimal_instance.type,
```

```
        current_cost=current_cost,
        optimal_cost=optimal_cost,
        monthly_savings=monthly_savings,
        cpu_waste_percent=cpu_waste * 100,
        memory_waste_percent=memory_waste * 100,
        risk_level=risk,
        confidence=calculate_confidence(metrics),
    )

def find_optimal_instance_type(
    required_cpu: float,
    required_memory: float,
    required_gpu: int,
    required_gpu_memory: float,
) -> InstanceType:
    """Find the smallest instance type that meets requirements"""

    # Get all available instance types
    available_types = get_available_instance_types()

    # Filter by GPU requirements
    candidates = [
        t for t in available_types
        if t.gpu_count >= required_gpu and
           t.gpu_memory >= required_gpu_memory
    ]

    # Filter by CPU/memory requirements
    candidates = [
        t for t in candidates
        if t.cpu >= required_cpu and
           t.memory >= required_memory
    ]

    # Sort by cost (cheapest first)
    candidates.sort(key=lambda t: t.hourly_rate)

    if not candidates:
        raise ValueError("No instance type meets requirements")

    return candidates[0]
```

LLM Prompt for Right-Sizing:

python

```
RIGHT_SIZING_PROMPT = """You are a cloud infrastructure optimization expert.

Current Instance:
- Type: {instance_type}
- vCPUs: {vcpus}, Memory: {memory}GB, GPUs: {gpus}
- Hourly Rate: ${hourly_rate}
- Monthly Cost: ${monthly_cost}

Utilization (Last 30 Days):
- CPU p95: {cpu_p95}% (waste: {cpu_waste}%)
- Memory p95: {memory_p95}% (waste: {memory_waste}%)
- GPU p95: {gpu_p95}%

Recommended Instance:
- Type: {recommended_type}
- vCPUs: {rec_vcpus}, Memory: {rec_memory}GB, GPUs: {rec_gpus}
- Hourly Rate: ${rec_hourly_rate}
- Monthly Cost: ${rec_monthly_cost}
- Estimated Savings: ${savings}/month ({savings_percent}%)

Historical Context:
{similar_right_sizings}

Your Task:
Analyze this right-sizing opportunity and provide a recommendation.

Consider:
1. Is the waste significant enough to justify the change?
2. Is the recommended instance appropriate (sufficient headroom)?
3. What are the risks of downsizing?
4. What safeguards should be in place?

Response format (JSON):
{
  "recommendation": "downsize" | "keep_current",
  "reasoning": "...",
  "confidence": 0.0-1.0,
  "risk_level": "low" | "medium" | "high",
  "implementation_steps": [...],
  "rollback_plan": "...",
  "estimated_savings": dollars
}
"""
```

6.3.3 Strategy 3: Reserved Instances

Concept: Purchase reserved instance commitments for predictable workloads (30-50% savings).

Algorithm:

python

```
def analyze_reserved_instance_opportunity(
    customer_id: str,
) -> ReservedInstanceAnalysis:
    """
    Analyze RI purchase opportunity

    Steps:
    1. Identify stable workloads (running >6 months continuously)
    2. Calculate baseline usage
    3. Compare on-demand vs RI costs
    4. Determine optimal RI term (1yr vs 3yr)
    5. Calculate break-even point
    """

    # Get all instances for customer
    instances = get_customer_instances(customer_id)

    # Identify stable instances (running >6 months)
    stable_instances = [
        inst for inst in instances
        if inst.uptime_days > 180 and
           inst.usage_pattern == "steady"
    ]

    # Group by instance type
    instance_groups = defaultdict(list)
    for inst in stable_instances:
        instance_groups[inst.type].append(inst)

    recommendations = []

    for instance_type, instances in instance_groups.items():
        # Calculate current on-demand cost
        on_demand_cost = sum(
            inst.hourly_rate * 730 for inst in instances
        )

        # Get RI pricing
        ri_1yr = get_ri_pricing(instance_type, term="1yr")
        ri_3yr = get_ri_pricing(instance_type, term="3yr")

        # Calculate RI costs
        ri_1yr_cost = ri_1yr.upfront + (ri_1yr.hourly * 730 * 12)
        ri_3yr_cost = ri_3yr.upfront + (ri_3yr.hourly * 730 * 36)

        # Annualize 3yr cost
        ri_3yr_annual = ri_3yr_cost / 3

        # Calculate savings
        savings_1yr = (on_demand_cost * 12) - ri_1yr_cost
        savings_3yr = (on_demand_cost * 12) - ri_3yr_annual

        # Break-even points
        breakeven_1yr = ri_1yr.upfront / (on_demand_cost - (ri_1yr.hourly * 730))
        breakeven_3yr = ri_3yr.upfront / (on_demand_cost - (ri_3yr.hourly * 730))
```

```
        # Recommendation
        if savings_3yr > savings_1yr:
            recommended_term = "3yr"
            savings = savings_3yr
        else:
            recommended_term = "1yr"
            savings = savings_1yr

        recommendations.append({
            "instance_type": instance_type,
            "instance_count": len(instances),
            "recommended_term": recommended_term,
            "annual_savings": savings,
            "breakeven_months": breakeven_3yr if recommended_term == "3yr" else breakeven_1yr,
            "confidence": 0.9,  # High confidence for stable workloads
        })

    return ReservedInstanceAnalysis(
        total_annual_savings=sum(r["annual_savings"] for r in recommendations),
        recommendations=recommendations,
    )
```

6.3.4 Strategy 4: Auto-Scaling Optimization

Concept: Optimize auto-scaling policies to match capacity with demand (15-25% savings).

Algorithm:

python

```python
def analyze_auto_scaling_opportunity(
    workload_id: str,
) -> AutoScalingAnalysis:
    """
    Analyze auto-scaling optimization opportunity

    Steps:
    1. Collect traffic patterns (90 days)
    2. Identify over-provisioning periods
    3. Analyze current scaling policy
    4. Recommend optimized scaling parameters
    5. Estimate cost savings
    """

    # Get traffic patterns
    traffic_data = get_traffic_patterns(
        workload_id=workload_id,
        days=90,
    )

    # Current scaling configuration
    current_config = get_auto_scaling_config(workload_id)

    # Analyze traffic patterns
    traffic_analysis = analyze_traffic(traffic_data)

    # Detect over-provisioning
    over_provisioned_hours = 0
    for hour in traffic_analysis.hourly:
        if hour.actual_instances > hour.required_instances:
            over_provisioned_hours += 1

    over_provision_percent = (over_provisioned_hours / len(traffic_analysis.hourly)) * 100

    # Calculate waste
    wasted_instance_hours = sum(
        max(0, hour.actual_instances - hour.required_instances)
        for hour in traffic_analysis.hourly
    )

    monthly_wasted_cost = wasted_instance_hours * current_config.instance_hourly_rate

    # Recommend optimized scaling policy
    optimized_policy = optimize_scaling_policy(
        traffic_patterns=traffic_analysis,
        current_policy=current_config,
    )

    # Estimate savings
    estimated_savings = monthly_wasted_cost * 0.7  # Conservative estimate

    return AutoScalingAnalysis(
        over_provision_percent=over_provision_percent,
        wasted_cost_per_month=monthly_wasted_cost,
        estimated_savings=estimated_savings,
```

```
        current_policy=current_config,
        recommended_policy=optimized_policy,
        confidence=0.8,
    )

def optimize_scaling_policy(
    traffic_patterns: TrafficAnalysis,
    current_policy: AutoScalingConfig,
) -> AutoScalingConfig:
    """Generate optimized auto-scaling policy"""

    # Calculate optimal thresholds
    # Use p75 as scale-up threshold (more aggressive)
    # Use p25 as scale-down threshold (more conservative)

    cpu_p75 = np.percentile(traffic_patterns.cpu_utilization, 75)
    cpu_p25 = np.percentile(traffic_patterns.cpu_utilization, 25)

    return AutoScalingConfig(
        min_instances=traffic_patterns.minimum_required,
        max_instances=traffic_patterns.peak_required + 2,  # Small buffer
        scale_up_threshold=min(cpu_p75, 70),  # Don't exceed 70%
        scale_down_threshold=max(cpu_p25, 30),  # Don't go below 30%
        scale_up_cooldown=60,  # seconds
        scale_down_cooldown=300,  # More conservative scale-down
        target_tracking_metric="RequestCountPerTarget",
        target_value=traffic_patterns.optimal_requests_per_instance,
    )
```

6.4 LangGraph State Machine

python

```python
from typing import TypedDict, List, Optional, Literal
from langgraph.graph import StateGraph, END

class CostAgentState(TypedDict):
    """State for Cost Optimization Agent"""
    # Workflow metadata
    workflow_id: str
    customer_id: str
    workload_id: str
    current_step: Literal[
        "init",
        "collect_metrics",
        "analyze",
        "recommend",
        "approval",
        "execute",
        "learn",
        "completed",
        "failed",
    ]

    # Metrics
    instance_metrics: dict
    cost_metrics: dict
    usage_patterns: dict

    # Analysis results
    spot_analysis: Optional[dict]
    right_sizing_analysis: Optional[dict]
    ri_analysis: Optional[dict]
    auto_scaling_analysis: Optional[dict]

    # Recommendations
    recommendations: List[dict]
    selected_recommendation: Optional[dict]

    # LLM interaction
    llm_prompts: List[str]
    llm_responses: List[dict]

    # Execution
    execution_plan: Optional[dict]
    execution_result: Optional[dict]

    # Learning
    outcome: Optional[dict]
    feedback_stored: bool

    # Error handling
    errors: List[str]
    retry_count: int

# Build workflow
def build_cost_agent_workflow() -> StateGraph:
    """Construct Cost Agent LangGraph workflow"""
```

```
workflow = StateGraph(CostAgentState)

# Add nodes
workflow.add_node("collect_metrics", collect_cost_metrics)
workflow.add_node("analyze_spot", analyze_spot_opportunities)
workflow.add_node("analyze_right_sizing", analyze_right_sizing)
workflow.add_node("analyze_ri", analyze_ri_opportunities)
workflow.add_node("analyze_auto_scaling", analyze_auto_scaling)
workflow.add_node("generate_recommendations", generate_cost_recommendations)
workflow.add_node("llm_reasoning", apply_llm_reasoning)
workflow.add_node("wait_approval", wait_for_approval)
workflow.add_node("execute_optimization", execute_cost_optimization)
workflow.add_node("monitor_execution", monitor_optimization_execution)
workflow.add_node("learn_outcome", learn_from_outcome)

# Define edges
workflow.add_edge("collect_metrics", "analyze_spot")
workflow.add_edge("analyze_spot", "analyze_right_sizing")
workflow.add_edge("analyze_right_sizing", "analyze_ri")
workflow.add_edge("analyze_ri", "analyze_auto_scaling")
workflow.add_edge("analyze_auto_scaling", "generate_recommendations")
workflow.add_edge("generate_recommendations", "llm_reasoning")

# Conditional: Auto-execute or wait for approval?
workflow.add_conditional_edges(
    "llm_reasoning",
    should_execute_automatically,
    {
        "execute": "execute_optimization",
        "approve": "wait_approval",
        "skip": END,
    }
)

workflow.add_edge("wait_approval", "execute_optimization")
workflow.add_edge("execute_optimization", "monitor_execution")

# Conditional: Success or rollback?
workflow.add_conditional_edges(
    "monitor_execution",
    check_execution_status,
    {
        "success": "learn_outcome",
        "rollback": "execute_rollback",
    }
)

workflow.add_edge("learn_outcome", END)

# Set entry point
workflow.set_entry_point("collect_metrics")

return workflow
```

6.5 Complete Data Models

6.5.1 PostgreSQL Schema

sql

```sql
-- Cost Agent Tables

CREATE TABLE cost_recommendations (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    customer_id UUID NOT NULL REFERENCES customers(id),
    workload_id UUID REFERENCES workloads(id),

    -- Recommendation details
    recommendation_type VARCHAR(50) NOT NULL, -- 'spot_migration', 'right_sizing', etc.
    status VARCHAR(20) NOT NULL DEFAULT 'pending', -- 'pending', 'approved', 'rejected', 'executed', 'rolled_back'
    confidence_score DECIMAL(3,2) NOT NULL CHECK (confidence_score BETWEEN 0 AND 1),
    risk_level VARCHAR(10) NOT NULL, -- 'low', 'medium', 'high'

    -- Impact estimates
    estimated_monthly_savings DECIMAL(10,2) NOT NULL,
    estimated_annual_savings DECIMAL(10,2) GENERATED ALWAYS AS (estimated_monthly_savings * 12) STORED,

    -- Current state
    current_config JSONB NOT NULL,

    -- Recommended state
    recommended_config JSONB NOT NULL,

    -- LLM reasoning
    llm_prompt TEXT,
    llm_response JSONB,
    reasoning TEXT,

    -- Approval
    requires_approval BOOLEAN DEFAULT true,
    approved_by UUID REFERENCES users(id),
    approved_at TIMESTAMP WITH TIME ZONE,

    -- Execution
    executed_at TIMESTAMP WITH TIME ZONE,
    execution_duration_seconds INTEGER,
    actual_savings DECIMAL(10,2),

    -- Metadata
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

    CONSTRAINT valid_status CHECK (
        status IN ('pending', 'approved', 'rejected', 'executing', 'executed', 'failed', 'rolled_back')
    )
);

CREATE INDEX idx_cost_recs_customer ON cost_recommendations(customer_id);
CREATE INDEX idx_cost_recs_status ON cost_recommendations(status);
CREATE INDEX idx_cost_recs_type ON cost_recommendations(recommendation_type);

-- Spot migration tracking
CREATE TABLE spot_migrations (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    recommendation_id UUID NOT NULL REFERENCES cost_recommendations(id),
```

```
    customer_id UUID NOT NULL REFERENCES customers(id),

    -- Original instance
    original_instance_id VARCHAR(100) NOT NULL,
    original_instance_type VARCHAR(50) NOT NULL,
    original_hourly_cost DECIMAL(8,4) NOT NULL,

    -- Spot instance
    spot_instance_id VARCHAR(100),
    spot_instance_type VARCHAR(50) NOT NULL,
    spot_hourly_cost DECIMAL(8,4) NOT NULL,
    max_spot_price DECIMAL(8,4) NOT NULL,

    -- Execution details
    status VARCHAR(20) NOT NULL DEFAULT 'pending',
    started_at TIMESTAMP WITH TIME ZONE,
    completed_at TIMESTAMP WITH TIME ZONE,

    -- Interruption tracking
    interruption_count INTEGER DEFAULT 0,
    last_interruption_at TIMESTAMP WITH TIME ZONE,

    -- Savings tracking
    actual_hourly_savings DECIMAL(8,4),
    total_savings_to_date DECIMAL(10,2),

    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Right-sizing tracking
CREATE TABLE right_sizing_changes (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    recommendation_id UUID NOT NULL REFERENCES cost_recommendations(id),
    customer_id UUID NOT NULL REFERENCES customers(id),

    -- Instance details
    instance_id VARCHAR(100) NOT NULL,
    old_instance_type VARCHAR(50) NOT NULL,
    new_instance_type VARCHAR(50) NOT NULL,

    -- Resource utilization (30-day metrics)
    cpu_p95_before DECIMAL(5,2),
    memory_p95_before DECIMAL(5,2),
    cpu_p95_after DECIMAL(5,2),
    memory_p95_after DECIMAL(5,2),

    -- Cost impact
    old_hourly_cost DECIMAL(8,4) NOT NULL,
    new_hourly_cost DECIMAL(8,4) NOT NULL,
    monthly_savings DECIMAL(10,2) NOT NULL,

    -- Execution
    status VARCHAR(20) NOT NULL DEFAULT 'pending',
    executed_at TIMESTAMP WITH TIME ZONE,
```

```
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Reserved instance recommendations
CREATE TABLE reserved_instance_recommendations (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    recommendation_id UUID NOT NULL REFERENCES cost_recommendations(id),
    customer_id UUID NOT NULL REFERENCES customers(id),

    -- RI details
    instance_type VARCHAR(50) NOT NULL,
    instance_count INTEGER NOT NULL,
    term VARCHAR(10) NOT NULL, -- '1yr' or '3yr'
    payment_option VARCHAR(20) NOT NULL, -- 'all_upfront', 'partial_upfront', 'no_upfront'

    -- Cost analysis
    on_demand_annual_cost DECIMAL(12,2) NOT NULL,
    ri_upfront_cost DECIMAL(12,2) NOT NULL,
    ri_annual_recurring_cost DECIMAL(12,2) NOT NULL,
    annual_savings DECIMAL(12,2) NOT NULL,
    breakeven_months DECIMAL(5,2) NOT NULL,

    -- Purchase status
    status VARCHAR(20) NOT NULL DEFAULT 'recommended',
    purchased_at TIMESTAMP WITH TIME ZONE,
    purchase_details JSONB,

    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Auto-scaling optimizations
CREATE TABLE auto_scaling_optimizations (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    recommendation_id UUID NOT NULL REFERENCES cost_recommendations(id),
    customer_id UUID NOT NULL REFERENCES customers(id),
    workload_id UUID NOT NULL REFERENCES workloads(id),

    -- Current policy
    current_min_instances INTEGER NOT NULL,
    current_max_instances INTEGER NOT NULL,
    current_scale_up_threshold DECIMAL(5,2),
    current_scale_down_threshold DECIMAL(5,2),

    -- Recommended policy
    recommended_min_instances INTEGER NOT NULL,
    recommended_max_instances INTEGER NOT NULL,
    recommended_scale_up_threshold DECIMAL(5,2),
    recommended_scale_down_threshold DECIMAL(5,2),

    -- Impact analysis
    over_provisioning_percent DECIMAL(5,2) NOT NULL,
    estimated_monthly_savings DECIMAL(10,2) NOT NULL,

    -- Execution
```

```
    status VARCHAR(20) NOT NULL DEFAULT 'pending',
    applied_at TIMESTAMP WITH TIME ZONE,

    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);
```

6.5.2 ClickHouse Schema (Metrics)

sql

```sql
-- Cost metrics time-series
CREATE TABLE cost_metrics (
    timestamp DateTime64(3),
    customer_id UUID,
    workload_id UUID,
    instance_id String,

    -- Cost data
    hourly_cost Decimal(8, 4),
    pricing_model String, -- 'on_demand', 'spot', 'reserved'

    -- Resource usage
    cpu_utilization Decimal(5, 2),
    memory_utilization Decimal(5, 2),
    gpu_utilization Decimal(5, 2),
    network_in_gb Decimal(10, 4),
    network_out_gb Decimal(10, 4),

    -- LLM-specific metrics
    tokens_processed UInt64,
    requests_processed UInt64,

    -- Cost efficiency metrics
    cost_per_token Decimal(10, 8),
    cost_per_request Decimal(8, 4),

    -- Metadata
    instance_type String,
    region String,
    availability_zone String,

    PRIMARY KEY (customer_id, timestamp)
)
ENGINE = MergeTree()
PARTITION BY toYYYYMM(timestamp)
ORDER BY (customer_id, workload_id, timestamp)
TTL timestamp + INTERVAL 90 DAY;

-- Spot interruption events
CREATE TABLE spot_interruption_events (
    timestamp DateTime64(3),
    customer_id UUID,
    instance_id String,
    instance_type String,
    region String,
    availability_zone String,
    interruption_reason String,
    uptime_hours Decimal(10, 2),

    PRIMARY KEY (customer_id, timestamp)
)
ENGINE = MergeTree()
PARTITION BY toYYYYMM(timestamp)
ORDER BY (customer_id, timestamp);
```

```
-- Savings tracking
CREATE TABLE cost_savings_daily (
    date Date,
    customer_id UUID,
    optimization_type String,

    -- Savings
    daily_savings Decimal(10, 2),
    cumulative_savings Decimal(12, 2),

    -- Breakdown
    spot_savings Decimal(10, 2),
    right_sizing_savings Decimal(10, 2),
    ri_savings Decimal(10, 2),
    auto_scaling_savings Decimal(10, 2),

    PRIMARY KEY (customer_id, date)
)
ENGINE = SummingMergeTree()
PARTITION BY toYYYYMM(date)
ORDER BY (customer_id, date);
```

6.6 APIs - All Endpoints

6.6.1 REST API Specification

python

```python
from fastapi import FastAPI, BackgroundTasks, HTTPException
from pydantic import BaseModel, Field
from typing import List, Optional

app = FastAPI(title="Cost Optimization Agent API")

# Request/Response Models
class OptimizationRequest(BaseModel):
    customer_id: str
    workload_id: Optional[str] = None
    optimization_types: List[str] = ["spot", "right_sizing", "ri", "auto_scaling"]

class OptimizationResponse(BaseModel):
    workflow_id: str
    status: str
    message: str

class RecommendationResponse(BaseModel):
    id: str
    type: str
    confidence: float
    estimated_savings: float
    risk_level: str
    reasoning: str
    requires_approval: bool

# Endpoints

@app.post("/optimize", response_model=OptimizationResponse)
async def trigger_optimization(
    request: OptimizationRequest,
    background_tasks: BackgroundTasks,
):
    """Trigger cost optimization workflow"""
    workflow_id = generate_workflow_id()

    # Start workflow in background
    background_tasks.add_task(
        run_cost_optimization_workflow,
        workflow_id=workflow_id,
        customer_id=request.customer_id,
        workload_id=request.workload_id,
        optimization_types=request.optimization_types,
    )

    return OptimizationResponse(
        workflow_id=workflow_id,
        status="started",
        message="Cost optimization workflow started",
    )

@app.get("/recommendations/{customer_id}", response_model=List[RecommendationResponse])
async def get_recommendations(
    customer_id: str,
    status: Optional[str] = None,
```

```python
    min_savings: Optional[float] = None,
):
    """Get cost optimization recommendations for customer"""
    query = db.query(CostRecommendation).filter(
        CostRecommendation.customer_id == customer_id
    )

    if status:
        query = query.filter(CostRecommendation.status == status)

    if min_savings:
        query = query.filter(CostRecommendation.estimated_monthly_savings >= min_savings)

    recommendations = query.all()

    return [
        RecommendationResponse(
            id=str(rec.id),
            type=rec.recommendation_type,
            confidence=float(rec.confidence_score),
            estimated_savings=float(rec.estimated_monthly_savings),
            risk_level=rec.risk_level,
            reasoning=rec.reasoning,
            requires_approval=rec.requires_approval,
        )
        for rec in recommendations
    ]

@app.post("/recommendations/{recommendation_id}/approve")
async def approve_recommendation(
    recommendation_id: str,
    approved_by: str,
):
    """Approve a cost optimization recommendation"""
    rec = db.query(CostRecommendation).filter_by(id=recommendation_id).first()

    if not rec:
        raise HTTPException(status_code=404, detail="Recommendation not found")

    rec.status = "approved"
    rec.approved_by = approved_by
    rec.approved_at = datetime.now()
    db.commit()

    # Trigger execution
    execute_cost_optimization(rec)

    return {"status": "approved", "execution_started": True}

@app.get("/savings/{customer_id}")
async def get_savings_summary(
    customer_id: str,
    start_date: Optional[date] = None,
    end_date: Optional[date] = None,
):
```

```
    """Get savings summary for customer"""
    if not start_date:
        start_date = date.today() - timedelta(days=30)
    if not end_date:
        end_date = date.today()

    # Query ClickHouse for savings data
    savings_data = clickhouse_client.query(f"""
        SELECT
            optimization_type,
            SUM(daily_savings) as total_savings,
            AVG(daily_savings) as avg_daily_savings
        FROM cost_savings_daily
        WHERE customer_id = '{customer_id}'
          AND date BETWEEN '{start_date}' AND '{end_date}'
        GROUP BY optimization_type
    """)

    return {
        "customer_id": customer_id,
        "period": {"start": start_date, "end": end_date},
        "total_savings": sum(row["total_savings"] for row in savings_data),
        "breakdown": savings_data,
    }

@app.get("/health")
async def health_check():
    """Health check endpoint"""
    return {
        "status": "healthy",
        "agent": "cost_optimization",
        "version": "1.0.0",
    }
```

6.7 Tools & Integrations

6.7.1 Cloud Provider APIs

python

```
class CloudProviderClient:
    """Abstract base class for cloud provider clients"""

    def get_instances(self, customer_id: str) -> List[Instance]:
        raise NotImplementedError

    def create_spot_request(self, spec: SpotRequestSpec) -> SpotRequest:
        raise NotImplementedError

    def terminate_instance(self, instance_id: str):
        raise NotImplementedError

    def modify_instance_type(self, instance_id: str, new_type: str):
        raise NotImplementedError

class AWSClient(CloudProviderClient):
    """AWS EC2 client"""

    def __init__(self, access_key: str, secret_key: str, region: str):
        self.ec2 = boto3.client(
            'ec2',
            aws_access_key_id=access_key,
            aws_secret_access_key=secret_key,
            region_name=region,
        )

    def get_instances(self, customer_id: str) -> List[Instance]:
        response = self.ec2.describe_instances(
            Filters=[
                {'Name': 'tag:customer_id', 'Values': [customer_id]},
                {'Name': 'instance-state-name', 'Values': ['running']},
            ]
        )

        instances = []
        for reservation in response['Reservations']:
            for inst in reservation['Instances']:
                instances.append(Instance(
                    id=inst['InstanceId'],
                    type=inst['InstanceType'],
                    state=inst['State']['Name'],
                    pricing_model='spot' if 'SpotInstanceRequestId' in inst else 'on_demand',
                    # ... more fields
                ))

        return instances

    def create_spot_request(self, spec: SpotRequestSpec) -> SpotRequest:
        response = self.ec2.request_spot_instances(
            InstanceCount=spec.count,
            LaunchSpecification={
                'ImageId': spec.ami_id,
                'InstanceType': spec.instance_type,
                'KeyName': spec.key_name,
                'SecurityGroupIds': spec.security_groups,
```

```
            'SubnetId': spec.subnet_id,
        },
        SpotPrice=str(spec.max_price),
    )

    return SpotRequest(
        request_id=response['SpotInstanceRequests'][0]['SpotInstanceRequestId'],
        status='pending',
    )
```

6.7.2 Pricing APIs

python

```
class PricingClient:
    """Client for cloud pricing data"""

    def get_on_demand_price(self, instance_type: str, region: str) -> float:
        """Get on-demand hourly price"""
        # AWS Pricing API
        pricing = boto3.client('pricing', region_name='us-east-1')

        response = pricing.get_products(
            ServiceCode='AmazonEC2',
            Filters=[
                {'Type': 'TERM_MATCH', 'Field': 'instanceType', 'Value': instance_type},
                {'Type': 'TERM_MATCH', 'Field': 'location', 'Value': region},
                {'Type': 'TERM_MATCH', 'Field': 'preInstalledSw', 'Value': 'NA'},
            ],
        )

        # Parse pricing data
        price_data = json.loads(response['PriceList'][0])
        on_demand_terms = price_data['terms']['OnDemand']
        price_dimensions = next(iter(on_demand_terms.values()))['priceDimensions']
        price = next(iter(price_dimensions.values()))['pricePerUnit']['USD']

        return float(price)

    def get_spot_price_history(
        self,
        instance_type: str,
        region: str,
        days: int = 90,
    ) -> List[SpotPricePoint]:
        """Get historical spot pricing"""
        ec2 = boto3.client('ec2', region_name=region)

        start_time = datetime.now() - timedelta(days=days)

        response = ec2.describe_spot_price_history(
            InstanceTypes=[instance_type],
            StartTime=start_time,
            ProductDescriptions=['Linux/UNIX'],
        )

        return [
            SpotPricePoint(
                timestamp=item['Timestamp'],
                price=float(item['SpotPrice']),
                availability_zone=item['AvailabilityZone'],
            )
            for item in response['SpotPriceHistory']
        ]
```

6.8 Error Handling & Rollback

python

```
class CostOptimizationError(Exception):
    """Base exception for cost optimization errors"""
    pass

class RollbackManager:
    """Manages rollback procedures for failed optimizations"""

    def __init__(self):
        self.rollback_handlers = {
            "spot_migration": self.rollback_spot_migration,
            "right_sizing": self.rollback_right_sizing,
            "auto_scaling": self.rollback_auto_scaling,
        }

    def rollback(self, recommendation: CostRecommendation) -> bool:
        """Execute rollback for failed optimization"""
        try:
            handler = self.rollback_handlers.get(recommendation.recommendation_type)

            if not handler:
                log.error(f"No rollback handler for {recommendation.recommendation_type}")
                return False

            log.info(f"Starting rollback for {recommendation.id}")
            result = handler(recommendation)

            if result:
                recommendation.status = "rolled_back"
                db.commit()
                log.info(f"Rollback successful for {recommendation.id}")

            return result

        except Exception as e:
            log.error(f"Rollback failed for {recommendation.id}: {e}")
            return False

    def rollback_spot_migration(self, recommendation: CostRecommendation) -> bool:
        """Rollback spot migration - restore on-demand instance"""
        migration = db.query(SpotMigration).filter_by(
            recommendation_id=recommendation.id
        ).first()

        if not migration:
            return False

        # Terminate spot instance
        cloud_client.terminate_instance(migration.spot_instance_id)

        # Launch on-demand replacement
        on_demand_instance = cloud_client.launch_instance(
            instance_type=migration.original_instance_type,
            config=recommendation.current_config,
        )
```

```
        # Wait for healthy
        if not wait_for_healthy(on_demand_instance, timeout=300):
            raise CostOptimizationError("Rollback failed - instance not healthy")

        # Restore traffic
        restore_traffic(on_demand_instance)

        return True

    def rollback_right_sizing(self, recommendation: CostRecommendation) -> bool:
        """Rollback right-sizing - restore original instance type"""
        change = db.query(RightSizingChange).filter_by(
            recommendation_id=recommendation.id
        ).first()

        # Modify instance type back to original
        cloud_client.modify_instance_type(
            instance_id=change.instance_id,
            new_type=change.old_instance_type,
        )

        return True
```

---

7. Performance Agent - DETAILED

7.1 Agent Overview

The **Performance Optimization Agent** ensures LLM inference workloads meet Service Level Objectives (SLOs) through intelligent scaling and optimization. It works closely with the Cost Agent to balance performance and cost.

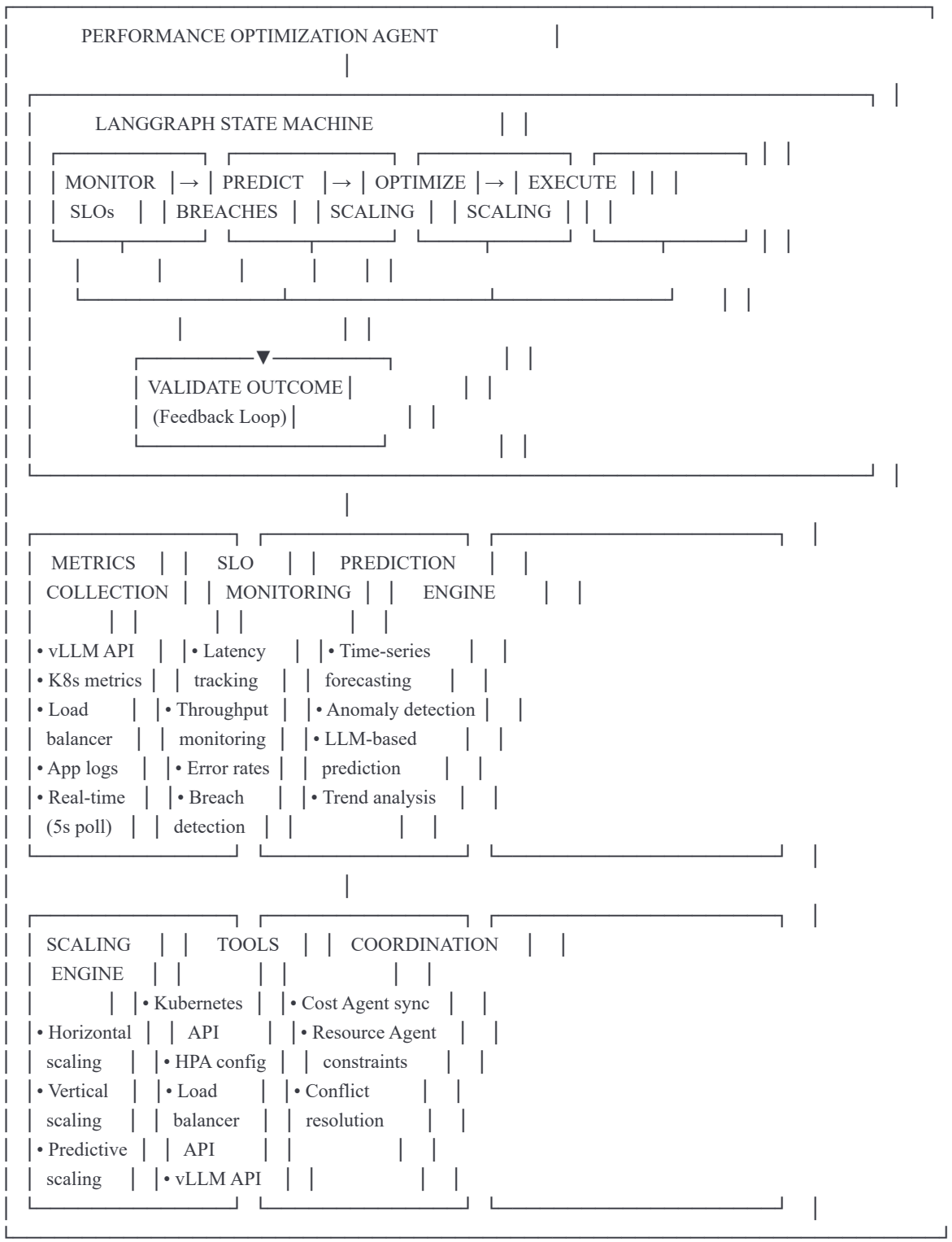
Primary Objectives:

1. Maintain 95%+ SLO compliance (latency, throughput, availability)
2. Predict and prevent SLO breaches before they occur
3. Optimize load balancing across instances
4. Coordinate with Cost Agent to avoid performance degradation during cost optimization

Key Metrics:

- **P95 Latency:** 95th percentile response time (target: <500ms)
- **Throughput:** Requests per second (tokens per second for LLM)
- **SLO Compliance:** % of requests meeting latency targets
- **Availability:** % uptime for inference endpoints

7.2 Complete Architecture



7.3 Five Optimization Strategies

7.3.1 Strategy 1: Predictive Scaling

Concept: Scale infrastructure before SLO breaches occur using time-series forecasting.

Algorithm:

python

```python
def predict_slo_breach(workload_id: str) -> PredictionResult:
    """
    Predict SLO breaches using time-series forecasting

    Steps:
    1. Collect historical metrics (last 7 days)
    2. Train time-series model (ARIMA or Prophet)
    3. Forecast next 2 hours
    4. Detect potential SLO breaches
    5. Generate pre-emptive scaling recommendation
    """

    # Get historical metrics
    metrics = get_performance_metrics(
        workload_id=workload_id,
        lookback_hours=168,  # 7 days
        metrics=["p95_latency", "throughput", "cpu_utilization"],
    )

    # Time-series forecasting
    forecast = forecast_metrics(
        historical_data=metrics,
        forecast_horizon_minutes=120,  # 2 hours
        model="prophet",  # or "arima"
    )

    # Detect potential breaches
    slo_target_latency = 500  # ms
    predicted_breaches = []

    for point in forecast:
        if point["p95_latency"] > slo_target_latency:
            predicted_breaches.append({
                "timestamp": point["timestamp"],
                "predicted_latency": point["p95_latency"],
                "confidence": point["confidence"],
            })

    if not predicted_breaches:
        return PredictionResult(
            breach_predicted=False,
            action_needed=False,
        )

    # Calculate required capacity
    current_instances = get_instance_count(workload_id)
    current_throughput = metrics[-1]["throughput"]
    predicted_throughput = forecast[0]["throughput"]

    required_instances = calculate_required_instances(
        current_instances=current_instances,
        current_throughput=current_throughput,
        predicted_throughput=predicted_throughput,
    )
```

```
    # Generate recommendation
    return PredictionResult(
        breach_predicted=True,
        first_breach_time=predicted_breaches[0]["timestamp"],
        action_needed=True,
        recommended_instances=required_instances,
        confidence=predicted_breaches[0]["confidence"],
        reasoning=f"Predicted latency breach at {predicted_breaches[0]['timestamp']}",
    )

def forecast_metrics(
    historical_data: pd.DataFrame,
    forecast_horizon_minutes: int,
    model: str = "prophet",
) -> List[dict]:
    """Forecast metrics using time-series model"""

    if model == "prophet":
        from prophet import Prophet

        # Prepare data for Prophet
        df = pd.DataFrame({
            'ds': historical_data['timestamp'],
            'y': historical_data['p95_latency'],
        })

        # Train model
        m = Prophet(
            changepoint_prior_scale=0.05,
            seasonality_mode='multiplicative',
        )
        m.add_seasonality(name='hourly', period=1/24, fourier_order=5)
        m.fit(df)

        # Generate forecast
        future = m.make_future_dataframe(periods=forecast_horizon_minutes, freq='T')
        forecast = m.predict(future)

        # Extract predictions
        predictions = []
        for idx, row in forecast.tail(forecast_horizon_minutes).iterrows():
            predictions.append({
                "timestamp": row['ds'],
                "p95_latency": row['yhat'],
                "lower_bound": row['yhat_lower'],
                "upper_bound": row['yhat_upper'],
                "confidence": 0.8,  # Prophet uncertainty interval
            })

        return predictions
```

7.3.2 Strategy 2: SLO-Aware Horizontal Scaling

Concept: Scale instances based on SLO compliance rather than just CPU/memory utilization.

Algorithm:

python

```python
def calculate_slo_aware_scaling(workload_id: str) -> ScalingRecommendation:
    """
    Calculate scaling based on SLO compliance

    Traditional HPA scales on CPU/memory.
    This scales on actual SLO metrics (latency, throughput).
    """

    # Get current SLO metrics
    current_metrics = get_current_slo_metrics(workload_id)

    # SLO targets
    slo_targets = {
        "p95_latency_ms": 500,
        "p99_latency_ms": 1000,
        "throughput_rps": 100,
        "error_rate_percent": 0.5,
    }

    # Calculate SLO compliance
    compliance = {
        "p95_latency": current_metrics["p95_latency"] <= slo_targets["p95_latency_ms"],
        "p99_latency": current_metrics["p99_latency"] <= slo_targets["p99_latency_ms"],
        "throughput": current_metrics["throughput"] >= slo_targets["throughput_rps"],
        "error_rate": current_metrics["error_rate"] <= slo_targets["error_rate_percent"],
    }

    overall_compliance = sum(compliance.values()) / len(compliance)

    # Determine scaling action
    if overall_compliance < 0.95:  # Not meeting SLOs
        # Calculate required scale-up
        latency_ratio = current_metrics["p95_latency"] / slo_targets["p95_latency_ms"]

        # More instances needed if latency too high
        if latency_ratio > 1.2:
            scale_factor = latency_ratio / 1.2
            return ScalingRecommendation(
                action="scale_up",
                scale_factor=scale_factor,
                reasoning=f"P95 latency {current_metrics['p95_latency']}ms exceeds target {slo_targets['p95_latency_ms']}ms",
                urgency="high",
            )

    elif overall_compliance > 0.98 and current_metrics["cpu_utilization"] < 40:
        # Over-provisioned - can scale down
        return ScalingRecommendation(
            action="scale_down",
            scale_factor=0.8,  # Reduce by 20%
            reasoning="SLO compliance >98% with CPU <40%, opportunity to reduce instances",
            urgency="low",
        )

    # No scaling needed
    return ScalingRecommendation(
```

```
        action="maintain",
        reasoning="SLO compliance satisfactory",
    )
```

7.3.3 Strategy 3: Load Balancing Optimization

Concept: Optimize load balancing algorithms to improve latency and throughput.

Algorithm:

python

```python
def optimize_load_balancing(workload_id: str) -> LoadBalancingRecommendation:
    """
    Analyze and optimize load balancing configuration

    Strategies:
    1. Least connections (better for long-running LLM requests)
    2. Weighted round-robin (favor more powerful instances)
    3. Adaptive (switch based on traffic patterns)
    """

    # Get current load balancing config
    current_config = get_load_balancer_config(workload_id)

    # Get per-instance metrics
    instance_metrics = get_per_instance_metrics(workload_id)

    # Analyze current distribution
    distribution_analysis = analyze_load_distribution(instance_metrics)

    # Check for imbalance
    if distribution_analysis["imbalance_score"] > 0.3:
        # Significant imbalance detected

        # Identify causes
        causes = []

        # Cause 1: Instance capacity variance
        capacity_variance = np.var([m["capacity"] for m in instance_metrics])
        if capacity_variance > 0.2:
            causes.append("Instance types have different capacities")

        # Cause 2: Request duration variance
        duration_variance = np.var([m["avg_request_duration"] for m in instance_metrics])
        if duration_variance > 0.3:
            causes.append("High variance in request durations")

        # Recommend strategy
        if "different capacities" in causes:
            return LoadBalancingRecommendation(
                strategy="weighted_round_robin",
                weights=calculate_instance_weights(instance_metrics),
                reasoning="Different instance capacities require weighted distribution",
            )

        elif "request durations" in causes:
            return LoadBalancingRecommendation(
                strategy="least_connections",
                reasoning="High request duration variance benefits from least connections",
            )

    # No change needed
    return LoadBalancingRecommendation(
        strategy="maintain_current",
        reasoning="Load distribution is balanced",
    )
```

```python
def calculate_instance_weights(instance_metrics: List[dict]) -> dict:
    """Calculate optimal weights for weighted round-robin"""

    weights = {}

    # Base weight on throughput capacity
    max_throughput = max(m["max_throughput"] for m in instance_metrics)

    for instance in instance_metrics:
        # Weight proportional to capacity
        weight = int((instance["max_throughput"] / max_throughput) * 100)
        weights[instance["instance_id"]] = weight

    return weights
```

7.3.4 Strategy 4: Latency Optimization

Concept: Reduce inference latency through batch size tuning and request routing.

Algorithm:


python

```python
def optimize_latency(workload_id: str) -> LatencyOptimizationResult:
    """
    Optimize latency through multiple techniques:
    1. Batch size tuning (vLLM)
    2. Request routing (short vs long context)
    3. KV cache warming
    """

    # Get current latency distribution
    latency_metrics = get_latency_metrics(workload_id, days=7)

    # Analyze latency by request characteristics
    latency_by_context_length = analyze_latency_by_context_length(latency_metrics)
    latency_by_batch_size = analyze_latency_by_batch_size(latency_metrics)

    optimizations = []

    # Optimization 1: Batch size tuning
    optimal_batch_size = find_optimal_batch_size(latency_by_batch_size)
    current_batch_size = get_current_batch_size(workload_id)

    if optimal_batch_size != current_batch_size:
        optimizations.append({
            "type": "batch_size_tuning",
            "current": current_batch_size,
            "optimal": optimal_batch_size,
            "expected_latency_reduction": estimate_latency_reduction(
                current_batch_size, optimal_batch_size
            ),
        })

    # Optimization 2: Request routing by context length
    # Long context requests → dedicated instances with more memory
    # Short context requests → high-throughput instances

    if should_enable_request_routing(latency_by_context_length):
        optimizations.append({
            "type": "request_routing",
            "strategy": "context_length_based",
            "thresholds": {
                "short_context": 2048,  # tokens
                "long_context": 8192,
            },
            "expected_latency_reduction": 0.15,  # 15% reduction
        })

    return LatencyOptimizationResult(
        optimizations=optimizations,
        estimated_latency_improvement=sum(
            opt.get("expected_latency_reduction", 0) for opt in optimizations
        ),
    )

def find_optimal_batch_size(latency_data: dict) -> int:
    """Find batch size that minimizes latency while maintaining throughput"""
```

```
    # Trade-off: Larger batches → better throughput but higher latency
    #            Smaller batches → lower latency but lower throughput

    optimal_batch_size = 1
    best_score = 0

    for batch_size, metrics in latency_data.items():
        # Score = throughput / latency
        # Higher is better
        score = metrics["throughput"] / metrics["p95_latency"]

        if score > best_score:
            best_score = score
            optimal_batch_size = batch_size

    return optimal_batch_size
```

7.3.5 Strategy 5: Throughput Optimization

Concept: Maximize requests/tokens per second through efficient resource utilization.

Algorithm:

python

```python
def optimize_throughput(workload_id: str) -> ThroughputOptimizationResult:
    """
    Optimize throughput through:
    1. Parallel request handling
    2. GPU utilization maximization
    3. Pipeline optimization
    """

    # Current throughput
    current_metrics = get_throughput_metrics(workload_id)
    current_throughput = current_metrics["requests_per_second"]
    current_tokens_per_second = current_metrics["tokens_per_second"]

    # GPU utilization
    gpu_utilization = current_metrics["gpu_utilization"]

    optimizations = []

    # Optimization 1: Increase parallel requests if GPU underutilized
    if gpu_utilization < 70:
        # GPU has capacity for more requests
        parallel_requests = current_metrics["parallel_requests"]
        optimal_parallel_requests = calculate_optimal_parallelism(
            gpu_utilization=gpu_utilization,
            current_parallelism=parallel_requests,
        )

        if optimal_parallel_requests > parallel_requests:
            optimizations.append({
                "type": "increase_parallelism",
                "current": parallel_requests,
                "optimal": optimal_parallel_requests,
                "expected_throughput_increase": 0.25,  # 25% increase
            })

    # Optimization 2: Tensor parallelism for large models
    model_size = get_model_size(workload_id)
    gpu_count = get_gpu_count(workload_id)

    if model_size > 70_000_000_000 and gpu_count > 1:  # >70B parameters, multi-GPU
        # Enable tensor parallelism
        optimizations.append({
            "type": "tensor_parallelism",
            "degree": min(gpu_count, 8),
            "expected_throughput_increase": 0.4,  # 40% increase
        })

    return ThroughputOptimizationResult(
        optimizations=optimizations,
        estimated_throughput_improvement=sum(
            opt.get("expected_throughput_increase", 0) for opt in optimizations
        ),
    )
```

7.4 LangGraph State Machine

python

```python
from typing import TypedDict, List, Optional, Literal
from langgraph.graph import StateGraph, END

class PerformanceAgentState(TypedDict):
    """State for Performance Optimization Agent"""
    workflow_id: str
    customer_id: str
    workload_id: str
    current_step: Literal[
        "init",
        "monitor_slos",
        "predict_breaches",
        "analyze_performance",
        "generate_optimization",
        "coordinate_with_cost",
        "execute_scaling",
        "validate_outcome",
        "completed",
        "failed",
    ]

    # SLO metrics
    current_slo_metrics: dict
    slo_targets: dict
    slo_compliance: float
    breach_detected: bool

    # Predictions
    predicted_breaches: List[dict]
    prediction_confidence: float

    # Analysis
    performance_analysis: dict
    bottlenecks: List[dict]

    # Optimization
    scaling_recommendation: Optional[dict]
    load_balancing_recommendation: Optional[dict]
    latency_optimization: Optional[dict]
    throughput_optimization: Optional[dict]

    # Coordination
    cost_agent_approval: Optional[bool]
    resource_constraints: Optional[dict]

    # Execution
    execution_result: Optional[dict]
    validation_result: Optional[dict]

    # LLM
    llm_prompts: List[str]
    llm_responses: List[dict]

    # Error handling
    errors: List[str]
```

```
    retry_count: int

def build_performance_agent_workflow() -> StateGraph:
    """Construct Performance Agent LangGraph workflow"""

    workflow = StateGraph(PerformanceAgentState)

    # Add nodes
    workflow.add_node("monitor_slos", monitor_slo_metrics)
    workflow.add_node("predict_breaches", predict_slo_breaches)
    workflow.add_node("analyze_performance", analyze_performance_bottlenecks)
    workflow.add_node("generate_scaling_rec", generate_scaling_recommendation)
    workflow.add_node("generate_latency_opt", generate_latency_optimization)
    workflow.add_node("generate_throughput_opt", generate_throughput_optimization)
    workflow.add_node("llm_reasoning", apply_llm_reasoning_performance)
    workflow.add_node("coordinate_cost_agent", coordinate_with_cost_agent)
    workflow.add_node("execute_optimization", execute_performance_optimization)
    workflow.add_node("validate_outcome", validate_performance_outcome)
    workflow.add_node("learn_outcome", learn_from_performance_outcome)

    # Define edges
    workflow.add_edge("monitor_slos", "predict_breaches")
    workflow.add_edge("predict_breaches", "analyze_performance")
    workflow.add_edge("analyze_performance", "generate_scaling_rec")
    workflow.add_edge("generate_scaling_rec", "generate_latency_opt")
    workflow.add_edge("generate_latency_opt", "generate_throughput_opt")
    workflow.add_edge("generate_throughput_opt", "llm_reasoning")

    # Conditional: Need Cost Agent coordination?
    workflow.add_conditional_edges(
        "llm_reasoning",
        need_cost_coordination,
        {
            "yes": "coordinate_cost_agent",
            "no": "execute_optimization",
        }
    )

    workflow.add_edge("coordinate_cost_agent", "execute_optimization")
    workflow.add_edge("execute_optimization", "validate_outcome")

    # Conditional: Validation success?
    workflow.add_conditional_edges(
        "validate_outcome",
        check_validation,
        {
            "success": "learn_outcome",
            "failed": "rollback_and_retry",
        }
    )

    workflow.add_edge("learn_outcome", END)

    workflow.set_entry_point("monitor_slos")
```

```
return workflow
```

7.5 Coordination with Cost Agent

python

```python
def coordinate_with_cost_agent(state: PerformanceAgentState) -> PerformanceAgentState:
    """
    Coordinate with Cost Agent to avoid conflicts

    Example conflict:
    - Performance Agent wants to scale up (SLO breach)
    - Cost Agent wants to scale down (save money)

    Resolution: Performance takes priority (SLO compliance critical)
    """

    # Check if Cost Agent has active optimization for same workload
    cost_optimization = get_active_cost_optimization(state["workload_id"])

    if not cost_optimization:
        # No conflict
        state["cost_agent_approval"] = True
        return state

    # Conflict detected
    conflict = {
        "performance_action": state["scaling_recommendation"]["action"],
        "cost_action": cost_optimization["action"],
        "conflict_type": "scale_up_vs_scale_down",
    }

    # Notify orchestrator
    resolution = orchestrator.resolve_conflict(
        agent_a="performance",
        agent_b="cost",
        conflict=conflict,
        priority_rules={"performance": 1, "cost": 3},  # Performance wins
    )

    if resolution["winner"] == "performance":
        # Cancel Cost Agent optimization
        orchestrator.cancel_optimization(cost_optimization["id"])
        state["cost_agent_approval"] = True
        state["llm_prompts"].append(
            f"Cost optimization {cost_optimization['id']} cancelled due to SLO priority"
        )
    else:
        # This shouldn't happen (Performance should always win SLO conflicts)
        state["cost_agent_approval"] = False
        state["errors"].append("SLO optimization overridden by cost optimization")

    return state
```

---

8. Resource Agent - DETAILED

8.1 Agent Overview

The **Resource Optimization Agent** focuses on maximizing GPU memory efficiency for LLM inference workloads. Its killer feature is **KVOptkit integration** - a tiered memory management system that enables 2-4x larger context windows by intelligently moving KV cache between GPU, CPU, and disk.

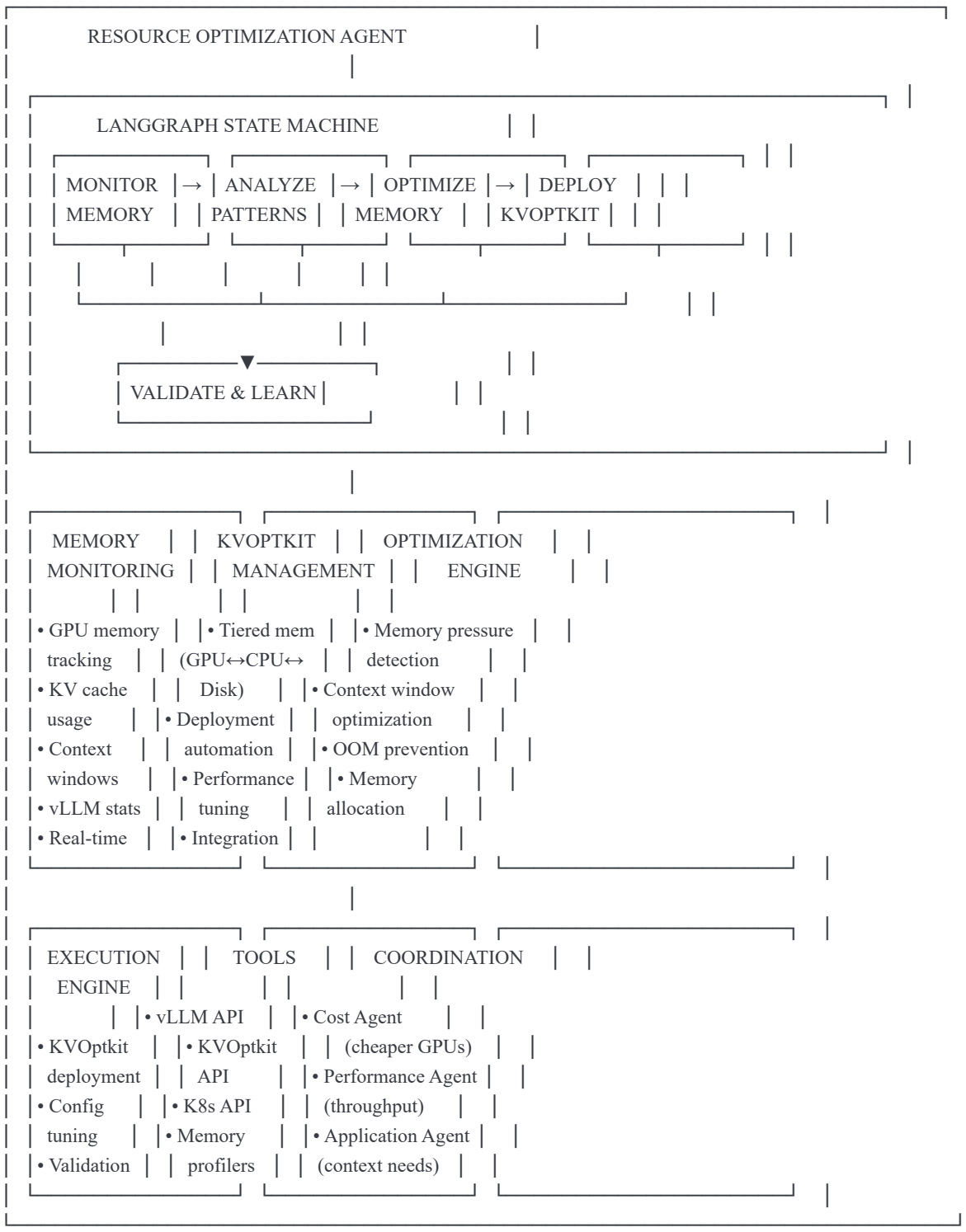
Primary Objectives:

1. Maximize effective context window size (2-4x increase)
2. Prevent OOM (Out of Memory) errors
3. Deploy and manage KVOptkit for tiered memory
4. Optimize memory allocation across workloads
5. Enable cost savings through better memory utilization

Impact:

- **2-4x larger context windows** without hardware upgrades
- **30-40% more throughput** per GPU
- **Enables cheaper GPU SKUs** (less VRAM needed)
- **Eliminates OOM crashes**

8.2 Complete Architecture



8.3 KVOptkit Technology Deep Dive

8.3.1 What is KVOptkit?

KVOptkit is a **tiered memory management system** for LLM inference that transparently moves KV cache between GPU memory, CPU memory, and disk storage based on access patterns and memory pressure.

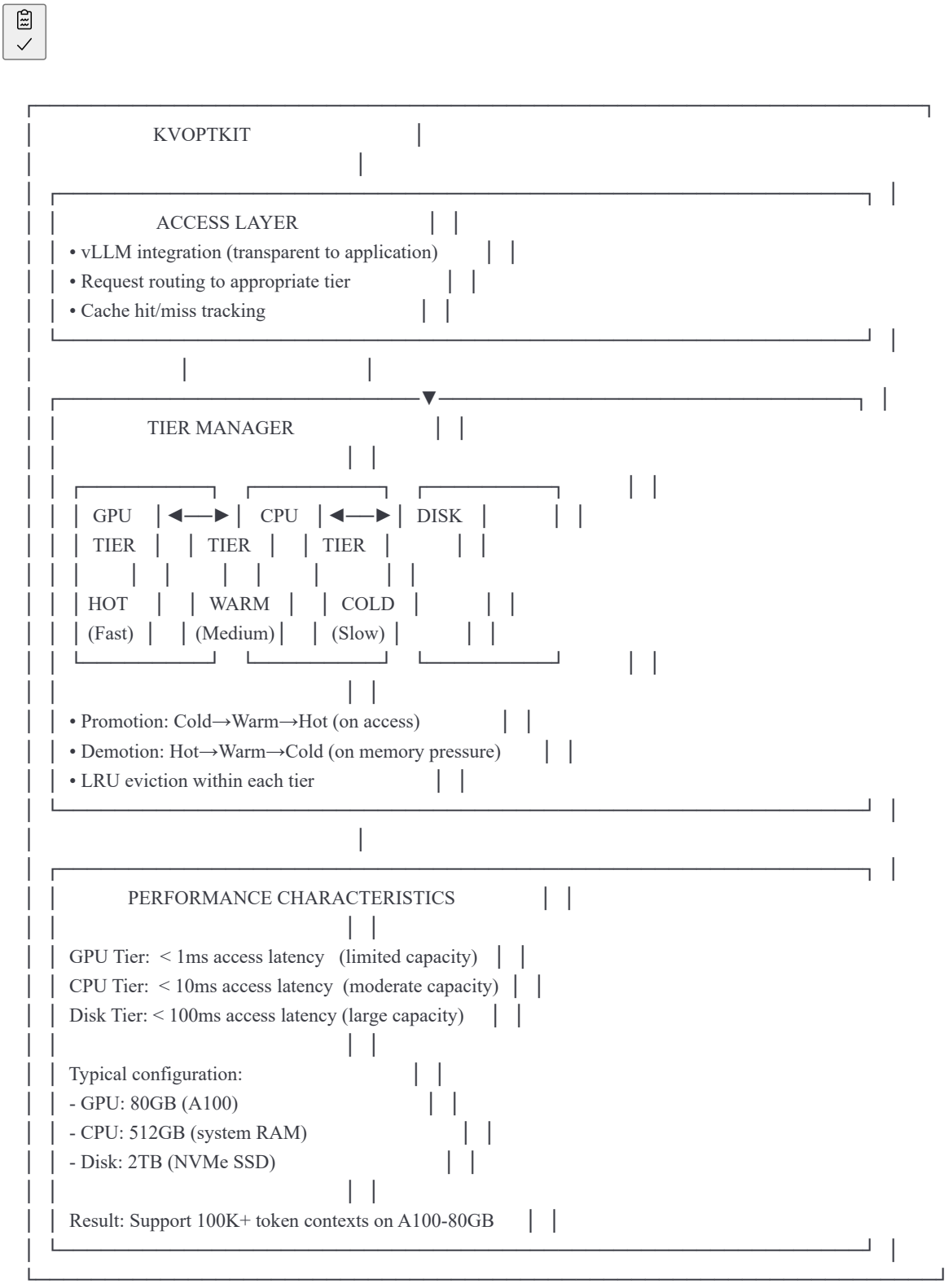
Problem it solves:

- GPU memory is expensive and limited
- Large context windows require huge KV cache
- Most KV cache entries are rarely accessed (long-tail distribution)
- Traditional approach: Either limit context size OR buy expensive high-memory GPUs

KVOptkit solution:

- **Hot data** → GPU memory (frequently accessed)
- **Warm data** → CPU memory (occasionally accessed)
- **Cold data** → Disk/NVMe (rarely accessed)
- **Transparent tiering** → Application doesn't know about tiers

8.3.2 Architecture



8.3.3 Integration with vLLM

python

```python
# KVOptkit vLLM integration

class KVOptKitEngine:
    """KVOptkit-enabled vLLM engine"""

    def __init__(
        self,
        model: str,
        gpu_memory_gb: int = 80,
        cpu_memory_gb: int = 512,
        disk_memory_gb: int = 2000,
    ):
        self.model = model

        # Initialize tier capacities
        self.tiers = {
            "gpu": TierConfig(
                capacity_gb=gpu_memory_gb,
                latency_ms=1,
                priority=1,  # Highest priority
            ),
            "cpu": TierConfig(
                capacity_gb=cpu_memory_gb,
                latency_ms=10,
                priority=2,
            ),
            "disk": TierConfig(
                capacity_gb=disk_memory_gb,
                latency_ms=100,
                priority=3,
            ),
        }

        # Initialize vLLM with KVOptkit
        self.engine = vLLM(
            model=model,
            kv_cache_manager=KVOptKitCacheManager(self.tiers),
            enable_tiered_memory=True,
        )

    def generate(self, prompt: str, max_tokens: int = 100) -> str:
        """Generate with automatic tier management"""

        # vLLM handles KV cache transparently
        # KVOptkit automatically moves cache between tiers
        output = self.engine.generate(
            prompt=prompt,
            max_tokens=max_tokens,
        )

        # Get tier statistics
        stats = self.engine.get_kv_cache_stats()

        return output, stats
```

```python
    # Tier statistics
class KVCacheStats:
    gpu_hit_rate: float      # % of accesses served from GPU
    cpu_hit_rate: float      # % served from CPU
    disk_hit_rate: float     # % served from disk

    gpu_utilization: float   # % of GPU tier full
    cpu_utilization: float   # % of CPU tier full
    disk_utilization: float  # % of disk tier full

    promotions: int          # Cold→Warm or Warm→Hot
    demotions: int           # Hot→Warm or Warm→Cold

    avg_access_latency_ms: float  # Average latency across all tiers
```

8.3.4 Performance Characteristics

python

```python
def estimate_kvoptkit_performance(
    context_length: int,
    model: str,
    gpu_memory_gb: int,
) -> KVOptKitPerformance:
    """Estimate performance with KVOptkit"""

    # Calculate KV cache size
    kv_cache_size_gb = calculate_kv_cache_size(
        model=model,
        context_length=context_length,
    )

    # Without KVOptkit: Limited by GPU memory
    max_context_without_kvoptkit = calculate_max_context(
        gpu_memory_gb=gpu_memory_gb,
        model=model,
    )

    # With KVOptkit: Use CPU + Disk tiers
    max_context_with_kvoptkit = calculate_max_context_tiered(
        gpu_memory_gb=gpu_memory_gb,
        cpu_memory_gb=512,
        disk_memory_gb=2000,
        model=model,
    )

    # Performance impact
    # GPU hit rate typically 80-90% (hot data)
    # Average latency increases ~10-20% due to tier misses
    gpu_hit_rate = 0.85
    avg_latency_increase = 0.15  # 15% slower on average

    return KVOptKitPerformance(
        max_context_without=max_context_without_kvoptkit,
        max_context_with=max_context_with_kvoptkit,
        context_increase_factor=max_context_with_kvoptkit / max_context_without_kvoptkit,
        estimated_latency_increase=avg_latency_increase,
        estimated_throughput_decrease=avg_latency_increase,  # Roughly equal
        gpu_hit_rate=gpu_hit_rate,
        memory_cost_reduction=0.4,  # Can use 40% cheaper GPUs (less VRAM needed)
    )

def calculate_kv_cache_size(model: str, context_length: int) -> float:
    """Calculate KV cache memory requirements"""

    # Model parameters
    model_configs = {
        "llama-70b": {
            "hidden_size": 8192,
            "num_layers": 80,
            "num_heads": 64,
        },
        "llama-13b": {
            "hidden_size": 5120,
```

```
        "num_layers": 40,
        "num_heads": 40,
    },
}

config = model_configs[model]

# KV cache size per token
# = 2 (K and V) * num_layers * hidden_size * bytes_per_param
bytes_per_param = 2  # FP16

kv_cache_per_token_bytes = (
    2 * config["num_layers"] * config["hidden_size"] * bytes_per_param
)

# Total for context
total_bytes = kv_cache_per_token_bytes * context_length
total_gb = total_bytes / (1024 ** 3)

return total_gb
```

8.4 Four Optimization Strategies

8.4.1 Strategy 1: Memory Pressure Detection & Resolution

python

```
def detect_memory_pressure(workload_id: str) -> MemoryPressureAnalysis:
    """
    Detect memory pressure before OOM occurs

    Leading indicators:
    1. GPU memory utilization > 90%
    2. Increasing swap usage (CPU tier)
    3. Rising allocation failure rate
    4. Context window truncation events
    """

    # Get memory metrics
    metrics = get_memory_metrics(workload_id, minutes=15)

    # Calculate pressure indicators
    gpu_util_p95 = np.percentile(metrics["gpu_memory_utilization"], 95)
    swap_usage = metrics["swap_usage_gb"][-1]
    allocation_failures = sum(metrics["allocation_failures"])
    truncation_events = sum(metrics["context_truncations"])

    # Pressure score (0-100)
    pressure_score = calculate_pressure_score(
        gpu_util=gpu_util_p95,
        swap_usage=swap_usage,
        failures=allocation_failures,
        truncations=truncation_events,
    )

    # Classify severity
    if pressure_score > 80:
        severity = "critical"  # OOM imminent
    elif pressure_score > 60:
        severity = "high"
    elif pressure_score > 40:
        severity = "medium"
    else:
        severity = "low"

    # Recommend actions
    actions = []

    if severity in ["critical", "high"]:
        # Immediate actions needed
        if not is_kvoptkit_enabled(workload_id):
            actions.append({
                "action": "deploy_kvoptkit",
                "urgency": "immediate",
                "expected_impact": "2-4x context window increase",
            })
        else:
            # KVOptkit already enabled, need more capacity
            actions.append({
                "action": "scale_up_instances",
                "urgency": "immediate",
                "expected_impact": "distribute memory load",
```

```
        })

    return MemoryPressureAnalysis(
        pressure_score=pressure_score,
        severity=severity,
        indicators={
            "gpu_utilization": gpu_util_p95,
            "swap_usage_gb": swap_usage,
            "allocation_failures": allocation_failures,
            "truncation_events": truncation_events,
        },
        recommended_actions=actions,
    )
```

8.4.2 Strategy 2: KVOptkit Deployment

python

```python
def plan_kvoptkit_deployment(workload_id: str) -> KVOptKitDeploymentPlan:
    """
    Plan KVOptkit deployment for workload

    Steps:
    1. Analyze current memory usage and patterns
    2. Determine optimal tier configuration
    3. Estimate performance impact
    4. Generate deployment plan
    """

    # Analyze current workload
    workload_analysis = analyze_workload_memory(workload_id)

    # Current state
    current_max_context = workload_analysis["max_context_length"]
    current_gpu_memory = workload_analysis["gpu_memory_gb"]
    avg_context_length = workload_analysis["avg_context_length"]

    # Determine if KVOptkit is beneficial
    # Beneficial if: avg context > 50% of max possible without KVOptkit
    benefit_threshold = 0.5
    context_ratio = avg_context_length / current_max_context

    if context_ratio < benefit_threshold:
        return KVOptKitDeploymentPlan(
            recommended=False,
            reason=f"Average context ({avg_context_length}) is only {context_ratio:.0%} of max ({current_max_context}). KVOptkit not needed.",
        )

    # KVOptkit is beneficial - plan deployment

    # Determine tier sizes
    # Rule of thumb:
    # - GPU tier: 80% of GPU memory for hot KV cache
    # - CPU tier: 8x GPU memory (if available)
    # - Disk tier: 32x GPU memory (cost-effective NVMe)

    tier_config = {
        "gpu_gb": current_gpu_memory * 0.8,
        "cpu_gb": min(current_gpu_memory * 8, 512),  # Limited by system RAM
        "disk_gb": current_gpu_memory * 32,
    }

    # Estimate new max context
    new_max_context = estimate_max_context_with_kvoptkit(
        gpu_gb=tier_config["gpu_gb"],
        cpu_gb=tier_config["cpu_gb"],
        disk_gb=tier_config["disk_gb"],
        model=workload_analysis["model"],
    )

    context_increase_factor = new_max_context / current_max_context

    # Performance impact estimate
```

```
performance_impact = estimate_kvoptkit_performance(
    context_length=new_max_context,
    model=workload_analysis["model"],
    gpu_memory_gb=current_gpu_memory,
)

# Generate deployment steps
deployment_steps = [
    {
        "step": 1,
        "action": "Provision NVMe storage (2TB) on instance",
        "duration_minutes": 10,
    },
    {
        "step": 2,
        "action": "Install KVOptkit library",
        "duration_minutes": 5,
    },
    {
        "step": 3,
        "action": "Update vLLM configuration to enable KVOptkit",
        "duration_minutes": 2,
    },
    {
        "step": 4,
        "action": "Restart vLLM service with new config",
        "duration_minutes": 5,
    },
    {
        "step": 5,
        "action": "Warm up KV cache (first requests slower)",
        "duration_minutes": 15,
    },
    {
        "step": 6,
        "action": "Monitor tier hit rates and performance",
        "duration_minutes": 30,
    },
]

return KVOptKitDeploymentPlan(
    recommended=True,
    tier_config=tier_config,
    current_max_context=current_max_context,
    new_max_context=new_max_context,
    context_increase_factor=context_increase_factor,
    estimated_latency_increase=performance_impact.estimated_latency_increase,
    estimated_gpu_hit_rate=performance_impact.gpu_hit_rate,
    deployment_steps=deployment_steps,
    total_deployment_time_minutes=sum(s["duration_minutes"] for s in deployment_steps),
)
```

8.4.3 Strategy 3: Context Window Optimization

python

```python
def optimize_context_windows(workload_id: str) -> ContextWindowOptimization:
    """
    Optimize context window sizes across requests

    Not all requests need maximum context.
    Right-size context per request type to save memory.
    """

    # Analyze actual context usage
    usage_analysis = analyze_context_usage(workload_id, days=30)

    # Group requests by type
    request_types = usage_analysis["request_types"]

    optimizations = []

    for req_type, stats in request_types.items():
        # Current: All requests get max context (wasteful)
        current_context = stats["allocated_context"]

        # Actual usage
        p95_actual_usage = np.percentile(stats["actual_context_used"], 95)

        # Recommended: p95 + 20% buffer
        recommended_context = int(p95_actual_usage * 1.2)

        # Memory savings
        memory_saved_per_request = (
            (current_context - recommended_context) *
            calculate_kv_cache_per_token(workload_analysis["model"])
        )

        requests_per_day = stats["count"]
        daily_memory_saved_gb = (
            memory_saved_per_request * requests_per_day / (1024 ** 3)
        )

        optimizations.append({
            "request_type": req_type,
            "current_context": current_context,
            "recommended_context": recommended_context,
            "reduction": current_context - recommended_context,
            "reduction_percent": ((current_context - recommended_context) / current_context) * 100,
            "daily_memory_saved_gb": daily_memory_saved_gb,
        })

    return ContextWindowOptimization(
        optimizations=optimizations,
        total_memory_saved_gb=sum(o["daily_memory_saved_gb"] for o in optimizations),
    )
```

8.4.4 Strategy 4: OOM Prevention

python

```python
def implement_oom_prevention(workload_id: str) -> OOMPreventionPlan:
    """
    Implement OOM prevention mechanisms

    Strategies:
    1. Request queuing when memory high
    2. Graceful degradation (reduce context)
    3. Circuit breaker (reject requests)
    4. Auto-scaling trigger on memory pressure
    """

    # Current OOM incidents
    oom_history = get_oom_incidents(workload_id, days=30)
    oom_count = len(oom_history)

    if oom_count == 0:
        # No OOM issues, minimal prevention needed
        return OOMPreventionPlan(
            prevention_needed=False,
            reason="No OOM incidents in last 30 days",
        )

    # Analyze OOM patterns
    oom_patterns = analyze_oom_patterns(oom_history)

    # Prevention strategies
    strategies = []

    # Strategy 1: Memory-based request queuing
    strategies.append({
        "strategy": "request_queuing",
        "trigger": "gpu_memory > 85%",
        "action": "Queue new requests until memory < 75%",
        "expected_effectiveness": 0.6,  # Prevents 60% of OOM
    })

    # Strategy 2: Dynamic context reduction
    if oom_patterns["cause"] == "large_context":
        strategies.append({
            "strategy": "dynamic_context_reduction",
            "trigger": "gpu_memory > 90%",
            "action": "Reduce context window by 20% for new requests",
            "expected_effectiveness": 0.3,
        })

    # Strategy 3: Circuit breaker
    strategies.append({
        "strategy": "circuit_breaker",
        "trigger": "3 OOM errors in 5 minutes",
        "action": "Reject requests with 503 Service Unavailable for 2 minutes",
        "expected_effectiveness": 1.0,  # Always prevents cascading OOM
    })

    # Strategy 4: Auto-scaling
    strategies.append({
```

```
        "strategy": "memory_based_autoscaling",
        "trigger": "gpu_memory > 80% for 5 minutes",
        "action": "Scale up +1 instance",
        "expected_effectiveness": 0.8,
    })

    return OOMPreventionPlan(
        prevention_needed=True,
        oom_incidents_last_30_days=oom_count,
        strategies=strategies,
        estimated_oom_reduction=sum(s["expected_effectiveness"] for s in strategies) / len(strategies),
    )
```

8.5 Data Models

8.5.1 PostgreSQL Schema

sql

```sql
-- Resource optimization tables

CREATE TABLE resource_optimizations (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    customer_id UUID NOT NULL REFERENCES customers(id),
    workload_id UUID REFERENCES workloads(id),

    optimization_type VARCHAR(50) NOT NULL,
    status VARCHAR(20) NOT NULL DEFAULT 'pending',

    -- Memory analysis
    current_gpu_memory_gb INTEGER NOT NULL,
    current_max_context INT NOT NULL,
    target_max_context INT,

    -- KVOptkit
    kvoptkit_enabled BOOLEAN DEFAULT false,
    tier_config JSONB,

    -- Impact
    estimated_context_increase_factor DECIMAL(4, 2),
    estimated_memory_savings_gb DECIMAL(8, 2),
    actual_context_increase_factor DECIMAL(4, 2),

    -- Execution
    deployed_at TIMESTAMP WITH TIME ZONE,

    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- KVOptkit deployments
CREATE TABLE kvoptkit_deployments (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    optimization_id UUID REFERENCES resource_optimizations(id),
    workload_id UUID NOT NULL REFERENCES workloads(id),

    -- Tier configuration
    gpu_tier_gb INTEGER NOT NULL,
    cpu_tier_gb INTEGER NOT NULL,
    disk_tier_gb INTEGER NOT NULL,

    -- Performance metrics
    gpu_hit_rate DECIMAL(5, 2),
    cpu_hit_rate DECIMAL(5, 2),
    disk_hit_rate DECIMAL(5, 2),
    avg_latency_increase_percent DECIMAL(5, 2),

    -- Status
    status VARCHAR(20) NOT NULL DEFAULT 'deploying',
    deployed_at TIMESTAMP WITH TIME ZONE,

    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Memory pressure events
```

```sql
CREATE TABLE memory_pressure_events (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    workload_id UUID NOT NULL REFERENCES workloads(id),

    timestamp TIMESTAMP WITH TIME ZONE NOT NULL,
    pressure_score INTEGER NOT NULL CHECK (pressure_score BETWEEN 0 AND 100),
    severity VARCHAR(20) NOT NULL,

    -- Indicators
    gpu_utilization_percent DECIMAL(5, 2),
    swap_usage_gb DECIMAL(8, 2),
    allocation_failures INTEGER,
    oom_errors INTEGER,

    -- Actions taken
    action_taken VARCHAR(100),
    action_result JSONB,

    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- OOM prevention rules
CREATE TABLE oom_prevention_rules (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    workload_id UUID NOT NULL REFERENCES workloads(id),

    rule_type VARCHAR(50) NOT NULL,
    trigger_condition JSONB NOT NULL,
    action JSONB NOT NULL,

    enabled BOOLEAN DEFAULT true,

    -- Effectiveness tracking
    times_triggered INTEGER DEFAULT 0,
    times_prevented_oom INTEGER DEFAULT 0,

    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);
```

8.5.2 ClickHouse Schema


sql

```
-- Memory metrics time-series
CREATE TABLE memory_metrics (
    timestamp DateTime64(3),
    customer_id UUID,
    workload_id UUID,
    instance_id String,

    -- GPU memory
    gpu_memory_used_gb Decimal(8, 2),
    gpu_memory_total_gb Decimal(8, 2),
    gpu_memory_utilization Decimal(5, 2),

    -- KV cache
    kv_cache_size_gb Decimal(8, 2),
    kv_cache_entries UInt64,

    -- KVOptkit tiers (if enabled)
    kvoptkit_enabled UInt8,
    gpu_tier_util Decimal(5, 2),
    cpu_tier_util Decimal(5, 2),
    disk_tier_util Decimal(5, 2),
    gpu_hit_rate Decimal(5, 2),
    cpu_hit_rate Decimal(5, 2),

    -- Context windows
    active_contexts UInt32,
    avg_context_length UInt32,
    max_context_length UInt32,

    PRIMARY KEY (customer_id, timestamp)
)
ENGINE = MergeTree()
PARTITION BY toYYYYMM(timestamp)
ORDER BY (customer_id, workload_id, timestamp)
TTL timestamp + INTERVAL 90 DAY;
```

8.6 Coordination with Other Agents

python

```python
def coordinate_resource_optimization(state: ResourceAgentState) -> ResourceAgentState:
    """
    Coordinate with Cost and Performance agents

    Example 1: Resource + Cost coordination
    - Resource Agent deploys KVOptkit
    - Enables 2x larger contexts on same GPU
    - Cost Agent can now use cheaper GPU SKU (less VRAM needed)

    Example 2: Resource + Performance coordination
    - Resource Agent detects memory pressure
    - Performance Agent adjusts scaling to compensate
    """

    optimization = state["selected_optimization"]

    # Notify Cost Agent of memory efficiency improvement
    if optimization["type"] == "kvoptkit_deployment":
        cost_impact = {
            "optimization_enabled": "cheaper_gpu_sku",
            "reason": "KVOptkit enables same context size on GPU with less VRAM",
            "potential_savings": estimate_gpu_cost_savings(
                current_sku=state["current_gpu_sku"],
                kvoptkit_enabled=True,
            ),
        }

        orchestrator.notify_agent(
            agent="cost",
            event_type="resource_optimization_completed",
            payload=cost_impact,
        )

    # Notify Performance Agent of throughput impact
    if optimization["type"] in ["kvoptkit_deployment", "context_window_optimization"]:
        performance_impact = {
            "throughput_change": optimization["estimated_throughput_impact"],
            "latency_change": optimization["estimated_latency_impact"],
            "reason": f"Resource optimization: {optimization['type']}",
        }

        orchestrator.notify_agent(
            agent="performance",
            event_type="resource_optimization_completed",
            payload=performance_impact,
        )

    return state
```

9. Application Agent - DETAILED

9.1 Agent Overview

The **Application Optimization Agent** is unique: it optimizes at the **application layer** (prompts, models, RAG pipelines) rather than infrastructure. It requires a **privacy-first** approach since it analyzes application behavior.

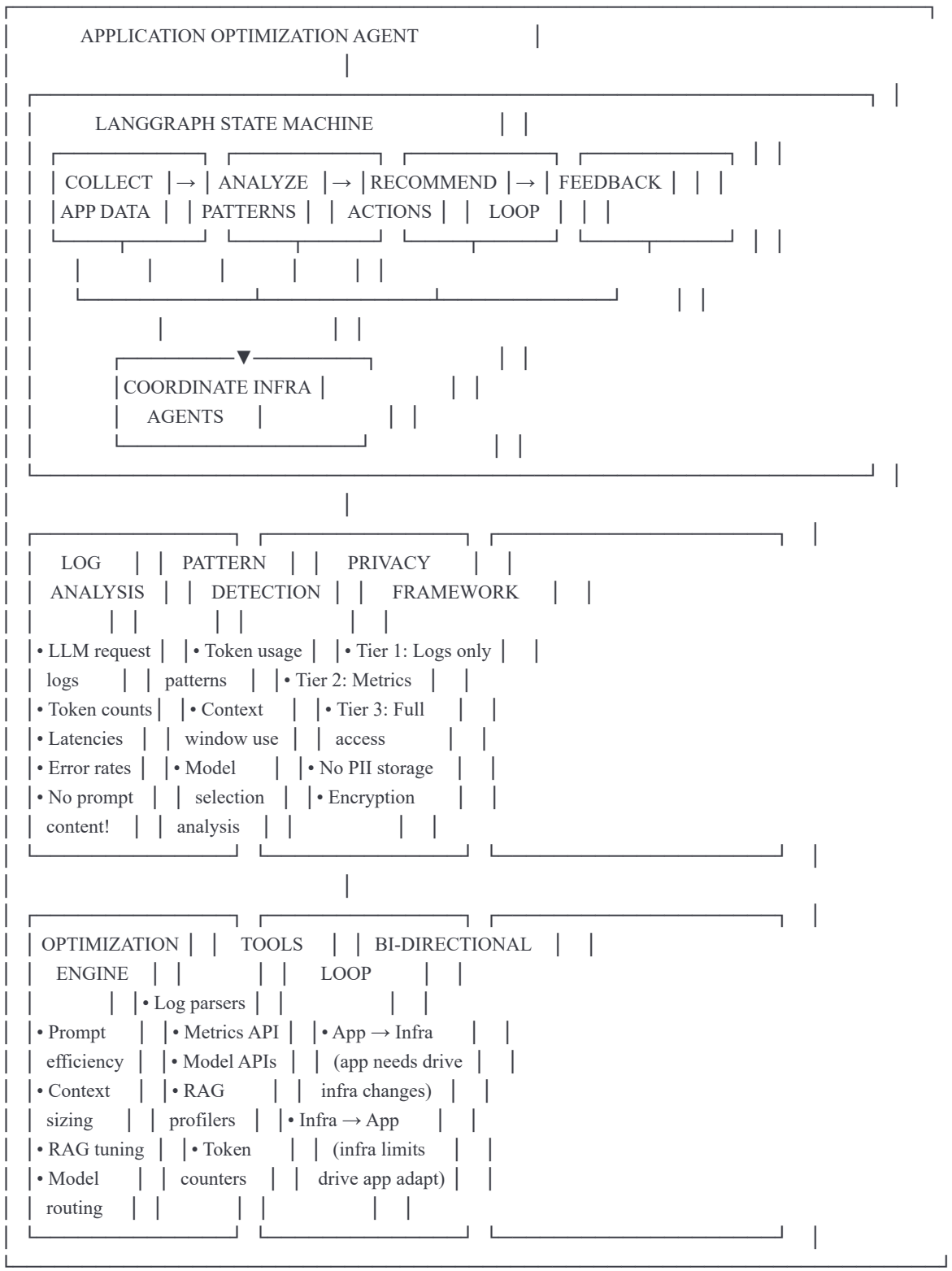
Primary Objectives:

1. Optimize prompt efficiency (reduce tokens, improve quality)
2. Right-size context windows per request type
3. Optimize RAG pipeline performance
4. Recommend optimal model routing
5. Create bi-directional optimization loop with infrastructure agents

Privacy Model:

- **Tier 1 (Default):** Log-based analysis only (no prompt content)
- **Tier 2 (Opt-in):** Metrics-based analysis (aggregated, anonymized)
- **Tier 3 (Full trust):** Deep integration with access to prompts (optional)

9.2 Complete Architecture



9.3 Privacy Model - Three Tiers

9.3.1 Tier 1: Log-Based Analysis (Default, No Sensitive Data)

What we collect:

- ✅ Request timestamps
- ✅ Token counts (input/output)
- ✅ Latency measurements
- ✅ Error rates
- ✅ Model used

- ✅ Context window size
- ❌ NO prompt content
- ❌ NO completions
- ❌ NO user IDs

python

```python
# Tier 1: Log-based analysis

class Tier1LogAnalyzer:
    """Privacy-preserving log analysis"""

    def analyze_logs(self, workload_id: str) -> Tier1Analysis:
        """Analyze logs without accessing sensitive data"""

        # Parse logs (structured format)
        logs = parse_llm_logs(workload_id, days=30)

        # Extract non-sensitive metrics
        metrics = {
            "total_requests": len(logs),
            "total_input_tokens": sum(log["input_tokens"] for log in logs),
            "total_output_tokens": sum(log["output_tokens"] for log in logs),
            "avg_latency_ms": np.mean([log["latency_ms"] for log in logs]),
            "error_rate": sum(1 for log in logs if log["error"]) / len(logs),
        }

        # Identify patterns (no prompt content needed)
        patterns = self.detect_patterns(logs)

        return Tier1Analysis(
            metrics=metrics,
            patterns=patterns,
            recommendations=self.generate_recommendations(patterns),
        )

    def detect_patterns(self, logs: List[dict]) -> dict:
        """Detect patterns from token counts and latencies"""

        # Pattern 1: Requests with very long inputs but short outputs
        # → Likely inefficient prompts or unnecessary context
        long_input_short_output = [
            log for log in logs
            if log["input_tokens"] > 5000 and log["output_tokens"] < 100
        ]

        # Pattern 2: High variance in latency for similar token counts
        # → Inconsistent performance, possible optimization opportunity
        latency_variance = np.var([log["latency_ms"] for log in logs])

        # Pattern 3: Frequent errors with specific token ranges
        # → Possible context window issues
        errors_by_tokens = defaultdict(int)
        for log in logs:
            if log["error"]:
                token_bucket = (log["input_tokens"] // 1000) * 1000
                errors_by_tokens[token_bucket] += 1

        return {
            "inefficient_prompts_count": len(long_input_short_output),
            "latency_variance": latency_variance,
```

```
        "errors_by_token_range": dict(errors_by_tokens),
    }
```

9.3.2 Tier 2: Metrics-Based Analysis (Opt-in, Aggregated Data)

What we collect:

- ✅ All Tier 1 data
- ✅ Aggregated prompt patterns (e.g., "50% of prompts use few-shot examples")
- ✅ RAG retrieval statistics (documents retrieved, relevance scores)
- ✅ Model performance comparisons
- ❌ Still NO actual prompt content
- ❌ NO user-identifiable data

python

```python
# Tier 2: Metrics-based analysis

class Tier2MetricsAnalyzer:
    """Metrics-based analysis with aggregation"""

    def analyze_metrics(self, workload_id: str) -> Tier2Analysis:
        """Analyze aggregated application metrics"""

        # Tier 1 analysis
        tier1 = Tier1LogAnalyzer().analyze_logs(workload_id)

        # Additional Tier 2 metrics (aggregated)
        metrics_api = get_application_metrics_api(workload_id)

        # RAG statistics (no document content)
        rag_stats = metrics_api.get_rag_statistics()

        # Model routing statistics
        model_usage = metrics_api.get_model_usage()

        # Prompt structure patterns (aggregated, no content)
        prompt_patterns = metrics_api.get_prompt_patterns()  # e.g., {"few_shot": 60%, "zero_shot": 40%}

        return Tier2Analysis(
            tier1_analysis=tier1,
            rag_performance=rag_stats,
            model_routing_efficiency=model_usage,
            prompt_structure_patterns=prompt_patterns,
            recommendations=self.generate_advanced_recommendations(
                tier1, rag_stats, model_usage
            ),
        )
```

9.3.3 Tier 3: Deep Integration (Full Trust, Full Access)

What we collect:

- ✅ All Tier 1 & 2 data
- ✅ Full prompt content (encrypted at rest)
- ✅ Completions

- ✅ User feedback
- ✅ Deep application insights

Use case: Customers who want maximum optimization and trust OptiInfra with sensitive data.


python

```python
# Tier 3: Deep integration

class Tier3DeepAnalyzer:
    """Deep analysis with full access (customer opt-in)"""

    def analyze_application(self, workload_id: str) -> Tier3Analysis:
        """Deep application analysis with prompt access"""

        # Lower tier analyses
        tier2 = Tier2MetricsAnalyzer().analyze_metrics(workload_id)

        # Access prompts (encrypted, audited)
        prompts = get_encrypted_prompts(workload_id, decrypt=True, audit=True)

        # Analyze prompt quality with LLM
        prompt_analysis = self.analyze_prompts_with_llm(prompts)

        # RAG pipeline deep analysis
        rag_pipeline = get_rag_pipeline_details(workload_id)
        rag_optimization = self.optimize_rag_pipeline(rag_pipeline)

        return Tier3Analysis(
            tier2_analysis=tier2,
            prompt_quality_analysis=prompt_analysis,
            rag_optimization=rag_optimization,
            recommendations=self.generate_deep_recommendations(
                tier2, prompt_analysis, rag_optimization
            ),
        )

    def analyze_prompts_with_llm(self, prompts: List[str]) -> PromptAnalysis:
        """Use LLM to analyze prompt quality"""

        # Sample prompts for analysis (not all)
        sample_prompts = random.sample(prompts, min(100, len(prompts)))

        analyses = []

        for prompt in sample_prompts:
            # Ask GPT-4 to analyze prompt
            analysis = llm_client.chat.completions.create(
                model="gpt-4o",
                messages=[
                    {"role": "system", "content": PROMPT_ANALYSIS_SYSTEM_PROMPT},
                    {"role": "user", "content": f"Analyze this prompt:\n\n{prompt}"},
                ],
            )

            analyses.append(json.loads(analysis.choices[0].message.content))

        # Aggregate insights
        return PromptAnalysis(
            avg_efficiency_score=np.mean([a["efficiency_score"] for a in analyses]),
            common_inefficiencies=[
                inefficiency
```

```
        for a in analyses
        for inefficiency in a["inefficiencies"]
    ],
    optimization_opportunities=aggregate_opportunities(analyses),
)
```

9.4 Five Optimization Strategies

9.4.1 Strategy 1: Prompt Efficiency Optimization

python

```python
def optimize_prompt_efficiency(workload_id: str) -> PromptOptimizationResult:
    """
    Optimize prompt efficiency to reduce tokens

    Techniques:
    1. Remove redundant instructions
    2. Optimize few-shot examples
    3. Use structured outputs
    4. Compress system prompts
    """

    # Tier 1: Token-based optimization (no prompt content needed)
    logs = get_llm_logs(workload_id, days=30)

    # Calculate token efficiency
    efficiency_metrics = calculate_token_efficiency(logs)

    # Identify inefficiencies
    inefficiencies = []

    # Inefficiency 1: Very long inputs with minimal outputs
    avg_input_tokens = np.mean([log["input_tokens"] for log in logs])
    avg_output_tokens = np.mean([log["output_tokens"] for log in logs])

    if avg_input_tokens > 3000 and avg_output_tokens < 500:
        token_waste = avg_input_tokens - (avg_output_tokens * 2)  # Rule of thumb
        inefficiencies.append({
            "type": "excessive_input_tokens",
            "waste_tokens_per_request": token_waste,
            "total_wasted_tokens_monthly": token_waste * len(logs) * 30 / 30,
            "recommendation": "Review prompts for unnecessary context or repetitive instructions",
        })

    # Inefficiency 2: High token variance
    token_variance = np.var([log["input_tokens"] for log in logs])
    if token_variance > 1000:
        inefficiencies.append({
            "type": "high_token_variance",
            "variance": token_variance,
            "recommendation": "Standardize prompt templates to reduce variance",
        })

    # Calculate potential savings
    if inefficiencies:
        potential_token_reduction = sum(
            ineff.get("waste_tokens_per_request", 0) for ineff in inefficiencies
        )

        monthly_cost_savings = calculate_token_cost_savings(
            tokens_saved_per_request=potential_token_reduction,
            requests_per_month=len(logs) * 30 / 30,
            model="gpt-4",
        )

        return PromptOptimizationResult(
```

```
            inefficiencies_found=len(inefficiencies),
            inefficiencies=inefficiencies,
            estimated_token_reduction_percent=20,  # Conservative estimate
            estimated_monthly_savings=monthly_cost_savings,
        )

    return PromptOptimizationResult(
        inefficiencies_found=0,
        message="Prompts are already efficient",
    )
```

9.4.2 Strategy 2: Context Window Right-Sizing

python

```python
def right_size_context_windows(workload_id: str) -> ContextRightSizingResult:
    """
    Right-size context windows per request type

    Many applications use max context for all requests (wasteful).
    Optimize context size based on actual usage.
    """

    # Analyze actual context usage
    logs = get_llm_logs(workload_id, days=30)

    # Group by request type (if available in logs)
    request_types = defaultdict(list)
    for log in logs:
        req_type = log.get("request_type", "default")
        request_types[req_type].append(log)

    optimizations = []

    for req_type, req_logs in request_types.items():
        # Current: Max context allocated
        current_context = max(log["input_tokens"] for log in req_logs)

        # Actual p95 usage
        p95_usage = np.percentile([log["input_tokens"] for log in req_logs], 95)

        # Recommended: p95 + 10% buffer
        recommended_context = int(p95_usage * 1.1)

        if recommended_context < current_context * 0.8:  # >20% savings
            memory_saved_per_request = calculate_memory_savings(
                current_context, recommended_context
            )

            optimizations.append({
                "request_type": req_type,
                "current_context": current_context,
                "recommended_context": recommended_context,
                "reduction_percent": ((current_context - recommended_context) / current_context) * 100,
                "memory_saved_mb": memory_saved_per_request,
                "requests_per_day": len(req_logs) * 30 / 30,
            })

    return ContextRightSizingResult(
        optimizations=optimizations,
        total_memory_saved_gb=sum(
            opt["memory_saved_mb"] * opt["requests_per_day"] / 1024
            for opt in optimizations
        ),
    )
```

9.4.3 Strategy 3: Model Routing Optimization

python

```python
def optimize_model_routing(workload_id: str) -> ModelRoutingOptimization:
    """
    Optimize which model handles which request type

    Not all requests need GPT-4. Route simple requests to cheaper models.
    """

    # Current model usage
    logs = get_llm_logs(workload_id, days=30)

    # Group by complexity (token counts, latency as proxies)
    simple_requests = [
        log for log in logs
        if log["input_tokens"] < 500 and log["output_tokens"] < 100
    ]

    complex_requests = [
        log for log in logs
        if log["input_tokens"] > 2000 or log["output_tokens"] > 500
    ]

    medium_requests = [
        log for log in logs
        if log not in simple_requests and log not in complex_requests
    ]

    # Current: All use GPT-4 (expensive)
    current_model = "gpt-4"
    current_cost_per_1k_tokens = 0.03  # Input tokens

    # Recommend routing
    routing_strategy = {
        "simple": {
            "model": "gpt-3.5-turbo",
            "cost_per_1k_tokens": 0.0015,
            "request_count": len(simple_requests),
        },
        "medium": {
            "model": "gpt-4-mini",
            "cost_per_1k_tokens": 0.015,
            "request_count": len(medium_requests),
        },
        "complex": {
            "model": "gpt-4",
            "cost_per_1k_tokens": 0.03,
            "request_count": len(complex_requests),
        },
    }

    # Calculate savings
    current_monthly_cost = sum(
        (log["input_tokens"] + log["output_tokens"]) / 1000 * current_cost_per_1k_tokens
        for log in logs
    ) * 30 / 30
```

```
    optimized_monthly_cost = 0
    for category, strategy in routing_strategy.items():
        category_logs = {"simple": simple_requests, "medium": medium_requests, "complex": complex_requests}[category]
        category_cost = sum(
            (log["input_tokens"] + log["output_tokens"]) / 1000 * strategy["cost_per_1k_tokens"]
            for log in category_logs
        ) * 30 / 30
        optimized_monthly_cost += category_cost

    monthly_savings = current_monthly_cost - optimized_monthly_cost

    return ModelRoutingOptimization(
        routing_strategy=routing_strategy,
        current_monthly_cost=current_monthly_cost,
        optimized_monthly_cost=optimized_monthly_cost,
        monthly_savings=monthly_savings,
        savings_percent=(monthly_savings / current_monthly_cost) * 100,
    )
```

9.4.4 Strategy 4: RAG Pipeline Optimization

python

```python
def optimize_rag_pipeline(workload_id: str) -> RAGOptimization:
    """
    Optimize RAG (Retrieval-Augmented Generation) pipeline

    Optimizations:
    1. Retrieval count (fewer documents if quality sufficient)
    2. Chunk size optimization
    3. Embedding model selection
    4. Re-ranking strategy
    """

    # Requires Tier 2 or Tier 3 access
    tier = get_privacy_tier(workload_id)

    if tier < 2:
        return RAGOptimization(
            optimization_available=False,
            reason="RAG optimization requires Tier 2 or Tier 3 privacy access",
        )

    # Get RAG metrics
    rag_metrics = get_rag_metrics(workload_id, days=30)

    optimizations = []

    # Optimization 1: Retrieval count
    avg_docs_retrieved = np.mean(rag_metrics["documents_retrieved"])
    avg_relevance_score = np.mean(rag_metrics["relevance_scores"])

    if avg_docs_retrieved > 10 and avg_relevance_score < 0.7:
        # Retrieving too many low-quality documents
        optimizations.append({
            "type": "reduce_retrieval_count",
            "current": avg_docs_retrieved,
            "recommended": 5,
            "reasoning": "High retrieval count with low relevance suggests noise",
        })

    # Optimization 2: Chunk size
    avg_chunk_size = np.mean(rag_metrics["chunk_sizes"])

    if avg_chunk_size > 1000:
        # Large chunks increase context usage
        optimizations.append({
            "type": "reduce_chunk_size",
            "current": avg_chunk_size,
            "recommended": 512,
            "reasoning": "Smaller chunks reduce context usage while maintaining quality",
        })

    return RAGOptimization(
        optimization_available=True,
        optimizations=optimizations,
        estimated_context_reduction=sum(
            opt.get("current", 0) - opt.get("recommended", 0)
```

```
            for opt in optimizations
        ),
    )
```

9.4.5 Strategy 5: Bi-Directional Optimization Loop

python

```python
def implement_bidirectional_loop(workload_id: str) -> BidirectionalLoop:
    """
    Implement bi-directional optimization between app and infrastructure

    Flow 1: Application → Infrastructure
    - App needs larger context windows
    - Triggers Resource Agent to deploy KVOptkit

    Flow 2: Infrastructure → Application
    - Infrastructure has memory constraints
    - Suggests Application Agent to optimize prompts/context
    """

    # Flow 1: App → Infra
    app_requirements = analyze_application_requirements(workload_id)

    if app_requirements["context_window_needed"] > app_requirements["context_window_available"]:
        # App needs more than infra can provide

        # Notify Resource Agent
        orchestrator.request_optimization(
            source_agent="application",
            target_agent="resource",
            request_type="increase_context_capacity",
            details={
                "current_capacity": app_requirements["context_window_available"],
                "required_capacity": app_requirements["context_window_needed"],
                "workload_id": workload_id,
            },
        )

        return BidirectionalLoop(
            flow="app_to_infra",
            action="requested_context_increase",
            details=app_requirements,
        )

    # Flow 2: Infra → App
    infra_constraints = get_infrastructure_constraints(workload_id)

    if infra_constraints["memory_pressure"] > 0.8:
        # Infrastructure under pressure

        # Suggest application optimizations
        app_optimizations = [
            "reduce_prompt_tokens",
            "optimize_context_windows",
            "improve_rag_efficiency",
        ]

        return BidirectionalLoop(
            flow="infra_to_app",
            action="suggested_app_optimizations",
            optimizations=app_optimizations,
            reason="Infrastructure memory pressure high",
```

```
    )

    return BidirectionalLoop(
        flow="balanced",
        message="Application and infrastructure in balance",
    )
```

9.5 Data Models

9.5.1 PostgreSQL Schema

sql

```sql
-- Application optimization tables

CREATE TABLE application_optimizations (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    customer_id UUID NOT NULL REFERENCES customers(id),
    workload_id UUID REFERENCES workloads(id),

    optimization_type VARCHAR(50) NOT NULL,
    privacy_tier INTEGER NOT NULL CHECK (privacy_tier IN (1, 2, 3)),
    status VARCHAR(20) NOT NULL DEFAULT 'pending',

    -- Analysis results (privacy-safe)
    token_efficiency_score DECIMAL(3, 2),
    context_waste_percent DECIMAL(5, 2),
    model_routing_efficiency DECIMAL(3, 2),

    -- Recommendations
    recommendations JSONB NOT NULL,
    estimated_token_savings_percent DECIMAL(5, 2),
    estimated_cost_savings DECIMAL(10, 2),

    -- Execution
    applied_at TIMESTAMP WITH TIME ZONE,
    actual_savings DECIMAL(10, 2),

    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Privacy tier tracking
CREATE TABLE privacy_tier_settings (
    customer_id UUID PRIMARY KEY REFERENCES customers(id),
    workload_id UUID REFERENCES workloads(id),

    privacy_tier INTEGER NOT NULL DEFAULT 1 CHECK (privacy_tier IN (1, 2, 3)),
    tier_description TEXT,

    -- Consent tracking
    opted_in_at TIMESTAMP WITH TIME ZONE,
    opted_in_by UUID REFERENCES users(id),

    -- Data access audit
    data_accessed_count INTEGER DEFAULT 0,
    last_data_access TIMESTAMP WITH TIME ZONE,

    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Prompt analysis (Tier 3 only, encrypted)
CREATE TABLE prompt_analyses (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    customer_id UUID NOT NULL REFERENCES customers(id),
    workload_id UUID REFERENCES workloads(id),

    -- Encrypted prompt (Tier 3 only)
    prompt_hash VARCHAR(64) NOT NULL,  -- SHA-256 hash
```

```
    prompt_encrypted BYTEA,  -- Encrypted prompt content

    -- Analysis results
    efficiency_score DECIMAL(3, 2),
    inefficiencies JSONB,
    optimization_suggestions JSONB,

    -- Audit
    analyzed_at TIMESTAMP WITH TIME ZONE NOT NULL,
    analyzed_by VARCHAR(100),

    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

    CONSTRAINT tier3_only CHECK (
        (SELECT privacy_tier FROM privacy_tier_settings WHERE customer_id = customer_analyses.customer_id) = 3
    )
);
```

---

10. Control Plane Services

10.1 Overview

The Control Plane provides the operational backbone for OptiInfra, handling API requests, background jobs, and centralized intelligence.

Components:

1. **API Gateway** - REST API for all operations
2. **Background Workers** - Async job processing (Celery)
3. **Intelligence Engine** - GPT-4o + RAG for reasoning

10.2 API Gateway

10.2.1 Architecture

python

```python
from fastapi import FastAPI, Depends, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials

app = FastAPI(
    title="OptiInfra Control Plane API",
    version="1.0.0",
    description="Central API for OptiInfra operations"
)

# CORS middleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=["https://portal.optiinfra.ai"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Authentication
security = HTTPBearer()

def verify_token(credentials: HTTPAuthorizationCredentials = Depends(security)):
    """Verify JWT token"""
    token = credentials.credentials
    # Verify token logic
    return decoded_user
```

10.2.2 Core Endpoints

python

```python
# Customer Management
@app.post("/customers")
async def create_customer(customer: CustomerCreate):
    """Create new customer"""
    pass

@app.get("/customers/{customer_id}")
async def get_customer(customer_id: str):
    """Get customer details"""
    pass

# Workload Management
@app.post("/workloads")
async def create_workload(workload: WorkloadCreate):
    """Register new workload"""
    pass

@app.get("/workloads/{workload_id}/status")
async def get_workload_status(workload_id: str):
    """Get workload status"""
    pass

# Optimization Requests
@app.post("/optimize")
async def trigger_optimization(request: OptimizationRequest):
    """Trigger optimization workflow"""
    # Route to appropriate agent(s) via orchestrator
    pass

@app.get("/optimizations/{optimization_id}")
async def get_optimization_status(optimization_id: str):
    """Get optimization status"""
    pass

# Recommendations
@app.get("/recommendations")
async def list_recommendations(
    customer_id: str,
    status: Optional[str] = None,
):
    """List recommendations"""
    pass

@app.post("/recommendations/{rec_id}/approve")
async def approve_recommendation(rec_id: str):
    """Approve recommendation for execution"""
    pass

# Reports & Analytics
@app.get("/reports/savings")
async def get_savings_report(customer_id: str, period: str):
    """Get cost savings report"""
    pass

@app.get("/reports/performance")
```

```python
async def get_performance_report(customer_id: str, period: str):
    """Get performance report"""
    pass
```

10.3 Background Workers

10.3.1 Celery Configuration

```python
from celery import Celery

celery_app = Celery(
    'optiinfra',
    broker='redis://valkey:6379/0',
    backend='redis://valkey:6379/0',
)

celery_app.conf.update(
    task_serializer='json',
    accept_content=['json'],
    result_serializer='json',
    timezone='UTC',
    enable_utc=True,
    task_track_started=True,
    task_time_limit=3600,  # 1 hour max
    worker_prefetch_multiplier=1,
)
```

10.3.2 Scheduled Tasks

```python
```

```python
from celery.schedules import crontab

celery_app.conf.beat_schedule = {
    # Collect metrics every 60 seconds
    'collect-metrics': {
        'task': 'tasks.collect_metrics',
        'schedule': 60.0,
    },

    # Run analysis every 5 minutes
    'run-analysis': {
        'task': 'tasks.run_periodic_analysis',
        'schedule': 300.0,
    },

    # Generate daily reports
    'generate-daily-reports': {
        'task': 'tasks.generate_daily_reports',
        'schedule': crontab(hour=6, minute=0),  # 6 AM daily
    },

    # Clean up old data
    'cleanup-old-data': {
        'task': 'tasks.cleanup_old_data',
        'schedule': crontab(hour=2, minute=0),  # 2 AM daily
    },
}
```

10.3.3 Task Definitions


python

```python
@celery_app.task
def collect_metrics(customer_id: str, workload_id: str):
    """Collect metrics from all sources"""
    # Trigger metrics collection across all agents
    orchestrator.broadcast_event(
        event_type="collect_metrics",
        payload={"customer_id": customer_id, "workload_id": workload_id},
    )

@celery_app.task
def run_periodic_analysis(customer_id: str):
    """Run periodic analysis for all workloads"""
    workloads = get_customer_workloads(customer_id)

    for workload in workloads:
        # Trigger analysis for each agent
        for agent_type in ["cost", "performance", "resource", "application"]:
            orchestrator.route_request(
                agent_type=agent_type,
                operation="analyze",
                payload={"workload_id": workload.id},
            )

@celery_app.task
def generate_daily_reports(customer_id: str):
    """Generate daily reports"""
    # Cost savings report
    cost_report = generate_cost_report(customer_id, period="daily")

    # Performance report
    perf_report = generate_performance_report(customer_id, period="daily")

    # Email to customer
    send_daily_report_email(customer_id, cost_report, perf_report)
```

10.4 Intelligence Engine

10.4.1 Architecture

python

```python
class IntelligenceEngine:
    """Centralized intelligence engine with GPT-4o + RAG"""

    def __init__(self):
        self.llm_client = OpenAI(api_key=settings.OPENAI_API_KEY)
        self.vector_db = QdrantClient(url=settings.QDRANT_URL)
        self.collection_name = "optimization_learnings"

    def query(
        self,
        prompt: str,
        context: Optional[dict] = None,
        use_rag: bool = True,
    ) -> IntelligenceResponse:
        """Query intelligence engine"""

        # Step 1: Retrieve relevant historical context (RAG)
        if use_rag and context:
            similar_cases = self.retrieve_similar_cases(context)
            rag_context = self.format_rag_context(similar_cases)
            augmented_prompt = f"{rag_context}\n\n{prompt}"
        else:
            augmented_prompt = prompt

        # Step 2: Query GPT-4o
        response = self.llm_client.chat.completions.create(
            model="gpt-4o",
            messages=[
                {"role": "system", "content": INTELLIGENCE_ENGINE_SYSTEM_PROMPT},
                {"role": "user", "content": augmented_prompt},
            ],
            temperature=0.7,
            max_tokens=2000,
        )

        # Step 3: Parse and validate response
        content = response.choices[0].message.content
        parsed = self.parse_llm_response(content)

        return IntelligenceResponse(
            reasoning=parsed["reasoning"],
            recommendation=parsed["recommendation"],
            confidence=parsed["confidence"],
            rag_cases_used=len(similar_cases) if use_rag else 0,
        )

    def retrieve_similar_cases(self, context: dict) -> List[dict]:
        """Retrieve similar historical cases from Qdrant"""

        # Create embedding of current context
        embedding = self.create_embedding(context)

        # Search Qdrant
        results = self.vector_db.search(
            collection_name=self.collection_name,
```

```
            query_vector=embedding,
            limit=5,
        )

        return [
            {
                "case_id": result.id,
                "similarity": result.score,
                "context": result.payload,
            }
            for result in results
        ]

    def store_outcome(self, optimization: dict, outcome: dict):
        """Store optimization outcome for future learning"""

        # Create embedding
        embedding = self.create_embedding(optimization)

        # Store in Qdrant
        self.vector_db.upsert(
            collection_name=self.collection_name,
            points=[{
                "id": str(uuid.uuid4()),
                "vector": embedding,
                "payload": {
                    "optimization": optimization,
                    "outcome": outcome,
                    "timestamp": datetime.now().isoformat(),
                },
            }],
        )
```

11. Customer Portal

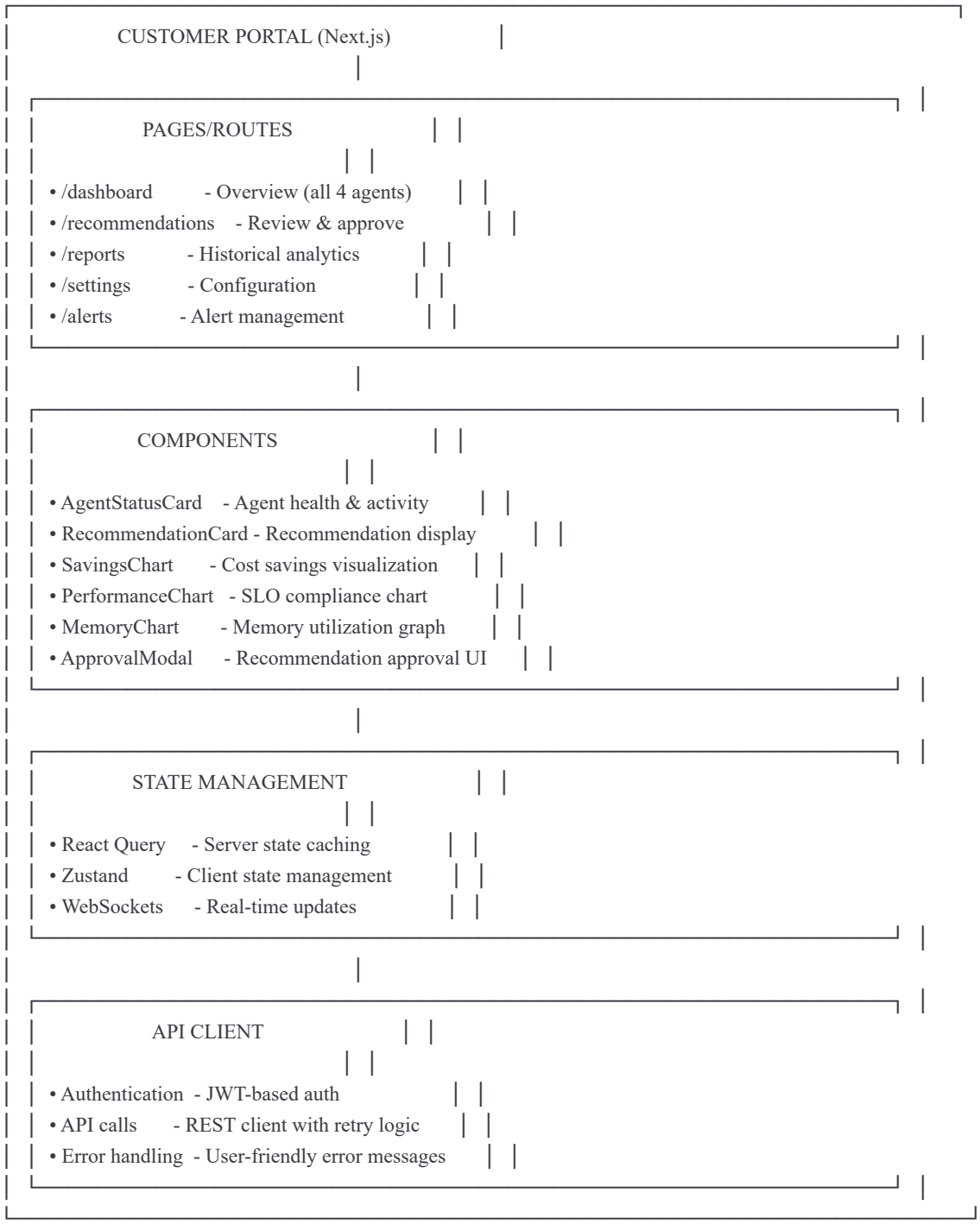
11.1 Overview

The Customer Portal is the primary interface for users to interact with OptiInfra. Built with **Next.js + React + Tailwind**, it provides real-time visibility into all four agents' activities.

Key Features:

- Real-time dashboard showing all agent activities
- Recommendation review and approval workflow
- Historical reports and analytics
- Configuration and settings
- Alert management

11.2 Portal Architecture



11.3 Dashboard View

typescript

```tsx
// Dashboard component showing all 4 agents

export default function Dashboard() {
  const { data: agents } = useQuery('agents', fetchAgentStatus);
  const { data: metrics } = useQuery('metrics', fetchMetrics);
  const { data: recommendations } = useQuery('recommendations', fetchRecommendations);

  return (
    <div className="container mx-auto p-6">
      <h1 className="text-3xl font-bold mb-6">OptiInfra Dashboard</h1>

      {/* Agent Status Cards */}
      <div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-4 gap-6 mb-8">
        <AgentStatusCard
          name="Cost Agent"
          status={agents?.cost?.status}
          lastRun={agents?.cost?.lastRun}
          savings="$12,450/month"
          icon={<DollarSignIcon />}
        />
        <AgentStatusCard
          name="Performance Agent"
          status={agents?.performance?.status}
          lastRun={agents?.performance?.lastRun}
          metric="98.5% SLO"
          icon={<TrendingUpIcon />}
        />
        <AgentStatusCard
          name="Resource Agent"
          status={agents?.resource?.status}
          lastRun={agents?.resource?.lastRun}
          metric="3.2x context increase"
          icon={<MemoryStickIcon />}
        />
        <AgentStatusCard
          name="Application Agent"
          status={agents?.application?.status}
          lastRun={agents?.application?.lastRun}
          metric="22% efficiency gain"
          icon={<CodeIcon />}
        />
      </div>

      {/* Savings Chart */}
      <div className="grid grid-cols-1 lg:grid-cols-2 gap-6 mb-8">
        <Card>
          <CardHeader>
            <CardTitle>Cost Savings (Last 30 Days)</CardTitle>
          </CardHeader>
          <CardContent>
            <SavingsChart data={metrics?.savings} />
          </CardContent>
        </Card>

        <Card>
```

```
        <CardHeader>
          <CardTitle>SLO Compliance</CardTitle>
        </CardHeader>
        <CardContent>
          <PerformanceChart data={metrics?.slo} />
        </CardContent>
      </Card>
    </div>

    {/* Recent Recommendations */}
    <Card>
      <CardHeader>
        <CardTitle>Pending Recommendations</CardTitle>
      </CardHeader>
      <CardContent>
        <RecommendationsList recommendations={recommendations} />
      </CardContent>
    </Card>
  </div>
 );
}
```

---

12. Data Architecture

12.1 Overview

OptiInfra uses a **multi-database architecture** optimized for different data types and access patterns.

Databases:

1. **PostgreSQL** - Transactional data, ACID compliance
2. **ClickHouse** - Time-series metrics, high throughput
3. **Valkey** - Caching, real-time state
4. **Qdrant** - Vector embeddings, semantic search

12.2 Database Selection Rationale

Database	Use Cases	Why Chosen
PostgreSQL	Customer data, recommendations, agent state, user accounts	ACID compliance, relational integrity, mature ecosystem
ClickHouse	Metrics (billions of data points), time-series analytics	100x faster than PostgreSQL for analytics, column-oriented, excellent compression
Valkey	Real-time caching, agent coordination, rate limiting	Sub-millisecond latency, Redis-compatible, open source
Qdrant	Historical learnings, RAG context, similar case retrieval	Purpose-built for vector search, high performance, easy integration

12.3 Complete Data Model

12.3.1 PostgreSQL Schema (Complete)

sql

```sql
-- ============================================================================
-- CORE TABLES
-- ============================================================================

-- Customers
CREATE TABLE customers (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    company VARCHAR(255),
    plan VARCHAR(50) DEFAULT 'standard',

    -- Cloud integration
    cloud_provider VARCHAR(50),
    cloud_credentials_encrypted BYTEA,

    -- Feature flags
    agents_enabled JSONB DEFAULT '["cost"]'::jsonb,
    operating_mode VARCHAR(20) DEFAULT 'advisory',

    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Workloads
CREATE TABLE workloads (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    customer_id UUID NOT NULL REFERENCES customers(id),
    name VARCHAR(255) NOT NULL,
    description TEXT,

    -- LLM configuration
    model VARCHAR(100) NOT NULL,
    vllm_endpoint VARCHAR(255),

    -- Infrastructure
    instance_ids TEXT[],
    region VARCHAR(50),

    -- Status
    status VARCHAR(20) DEFAULT 'active',

    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Agents
CREATE TABLE agents (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    agent_type VARCHAR(50) NOT NULL,
    version VARCHAR(20) NOT NULL,
    endpoint VARCHAR(255) NOT NULL,
    status VARCHAR(20) DEFAULT 'healthy',

    last_heartbeat TIMESTAMP WITH TIME ZONE,
```

```
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- ============================================================================
-- RECOMMENDATION TABLES (All Agents)
-- ============================================================================

-- Base recommendations table
CREATE TABLE recommendations (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    customer_id UUID NOT NULL REFERENCES customers(id),
    workload_id UUID REFERENCES workloads(id),
    agent_type VARCHAR(50) NOT NULL,

    recommendation_type VARCHAR(100) NOT NULL,
    status VARCHAR(20) DEFAULT 'pending',

    -- Recommendation details
    title VARCHAR(255) NOT NULL,
    description TEXT NOT NULL,
    reasoning TEXT,

    -- Impact estimates
    estimated_impact JSONB NOT NULL,
    confidence_score DECIMAL(3,2) CHECK (confidence_score BETWEEN 0 AND 1),
    risk_level VARCHAR(10),

    -- Approval workflow
    requires_approval BOOLEAN DEFAULT true,
    approved_by UUID REFERENCES users(id),
    approved_at TIMESTAMP WITH TIME ZONE,
    rejected_by UUID REFERENCES users(id),
    rejected_at TIMESTAMP WITH TIME ZONE,
    rejection_reason TEXT,

    -- Execution
    executed_at TIMESTAMP WITH TIME ZONE,
    execution_result JSONB,
    actual_impact JSONB,

    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- ============================================================================
-- AUDIT & HISTORY
-- ============================================================================

-- Audit log
CREATE TABLE audit_log (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    customer_id UUID REFERENCES customers(id),
    user_id UUID REFERENCES users(id),
    agent_type VARCHAR(50),
```

```
    action VARCHAR(100) NOT NULL,
    entity_type VARCHAR(50),
    entity_id UUID,

    details JSONB,

    ip_address INET,
    user_agent TEXT,

    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE INDEX idx_audit_log_customer ON audit_log(customer_id);
CREATE INDEX idx_audit_log_created ON audit_log(created_at);
```

12.3.2 ClickHouse Schema (Complete)

sql

```
-- ============================================================================
-- METRICS TABLES (Time-Series)
-- ============================================================================

-- Unified metrics table
CREATE TABLE metrics (
    timestamp DateTime64(3),
    customer_id UUID,
    workload_id UUID,
    instance_id String,
    metric_type String,

    -- Cost metrics
    hourly_cost Decimal(8, 4),
    cumulative_cost Decimal(12, 2),

    -- Performance metrics
    p50_latency_ms Decimal(10, 2),
    p95_latency_ms Decimal(10, 2),
    p99_latency_ms Decimal(10, 2),
    throughput_rps Decimal(10, 2),
    error_rate Decimal(5, 4),

    -- Resource metrics
    cpu_utilization Decimal(5, 2),
    memory_utilization Decimal(5, 2),
    gpu_utilization Decimal(5, 2),
    gpu_memory_used_gb Decimal(8, 2),

    -- LLM-specific metrics
    tokens_processed UInt64,
    requests_processed UInt64,
    active_contexts UInt32,
    avg_context_length UInt32,

    PRIMARY KEY (customer_id, workload_id, timestamp)
)
ENGINE = MergeTree()
PARTITION BY toYYYYMM(timestamp)
ORDER BY (customer_id, workload_id, instance_id, timestamp)
TTL timestamp + INTERVAL 90 DAY;

-- ============================================================================
-- AGGREGATED VIEWS (Materialized)
-- ============================================================================

-- Daily aggregations
CREATE MATERIALIZED VIEW metrics_daily
ENGINE = SummingMergeTree()
PARTITION BY toYYYYMM(date)
ORDER BY (customer_id, workload_id, date)
AS SELECT
    toDate(timestamp) AS date,
    customer_id,
    workload_id,
```

```
    -- Cost aggregations
    sum(hourly_cost) AS total_cost,
    avg(hourly_cost) AS avg_hourly_cost,

    -- Performance aggregations
    avg(p95_latency_ms) AS avg_p95_latency,
    max(p95_latency_ms) AS max_p95_latency,
    avg(throughput_rps) AS avg_throughput,
    avg(error_rate) AS avg_error_rate,

    -- Resource aggregations
    avg(cpu_utilization) AS avg_cpu,
    avg(memory_utilization) AS avg_memory,
    avg(gpu_utilization) AS avg_gpu,

    -- Volume metrics
    sum(tokens_processed) AS total_tokens,
    sum(requests_processed) AS total_requests
FROM metrics
GROUP BY date, customer_id, workload_id;
```

12.3.3 Valkey (Redis) Data Structures

python

```
# Valkey key patterns and data structures

# 1. Agent registry (Hash)
KEY: "agent:registry:{agent_id}"
VALUE: {
  "agent_type": "cost",
  "version": "1.0.0",
  "endpoint": "http://cost-agent:8001",
  "status": "healthy",
  "last_heartbeat": "2025-10-10T12:00:00Z",
}
TTL: 60 seconds (auto-expires if heartbeat stops)

# 2. Agent state snapshots (Hash)
KEY: "agent:state:{agent_id}:{workflow_id}"
VALUE: {
  "workflow_id": "wf_123",
  "current_step": "analyze",
  "state": {...},
  "updated_at": "2025-10-10T12:00:00Z",
}
TTL: 24 hours

# 3. Shared state (String, JSON)
KEY: "state:{customer_id}:{workload_id}"
VALUE: JSON of SharedState
TTL: 24 hours

# 4. Rate limiting (Counter)
KEY: "ratelimit:{customer_id}:{operation}"
VALUE: request_count
TTL: 3600 seconds (1 hour window)

# 5. Real-time metrics cache (Hash)
KEY: "metrics:realtime:{workload_id}"
VALUE: {
  "cpu": 0.75,
  "memory": 0.60,
  "latency_p95": 450,
  "updated_at": "2025-10-10T12:00:00Z",
}
TTL: 60 seconds

# 6. Event pub/sub (Pub/Sub channels)
CHANNEL: "events:optimization_completed"
CHANNEL: "events:slo_breach"
CHANNEL: "events:recommendation_generated"
```

12.3.4 Qdrant Collections

python

```
# Qdrant vector database collections

# Collection: optimization_learnings
{
  "collection_name": "optimization_learnings",
  "vector_size": 1536,  # OpenAI embedding size
  "distance": "Cosine",
  "payload_schema": {
    "optimization_type": "string",
    "agent_type": "string",
    "customer_id": "string",
    "context": {
      "instance_type": "string",
      "region": "string",
      "workload_type": "string",
      "current_cost": "float",
      "current_performance": "dict",
    },
    "action_taken": "dict",
    "outcome": {
      "success": "bool",
      "actual_savings": "float",
      "actual_performance_change": "dict",
    },
    "timestamp": "datetime",
  },
}

# Collection: prompt_embeddings (Tier 3 only)
{
  "collection_name": "prompt_embeddings",
  "vector_size": 1536,
  "distance": "Cosine",
  "payload_schema": {
    "customer_id": "string",
    "prompt_hash": "string",
    "prompt_type": "string",
    "token_count": "int",
    "efficiency_score": "float",
    "optimization_applied": "dict",
    "timestamp": "datetime",
  },
}
```

12.4 Data Flow Patterns

12.4.1 Metrics Collection Flow

1. vLLM/Cloud APIs expose metrics
2. Agent collectors poll every 60s
3. Raw metrics → ClickHouse (high throughput)
4. Aggregated metrics → Valkey (real-time cache)
5. Agents query Valkey for latest metrics
6. Historical analysis queries ClickHouse

12.4.2 Recommendation Flow

1. Agent generates recommendation
2. Recommendation → PostgreSQL (transactional)
3. Event published → Valkey pub/sub
4. Portal receives real-time update via WebSocket
5. User approves → PostgreSQL updated
6. Execution triggered
7. Outcome stored → PostgreSQL + Qdrant (learning)

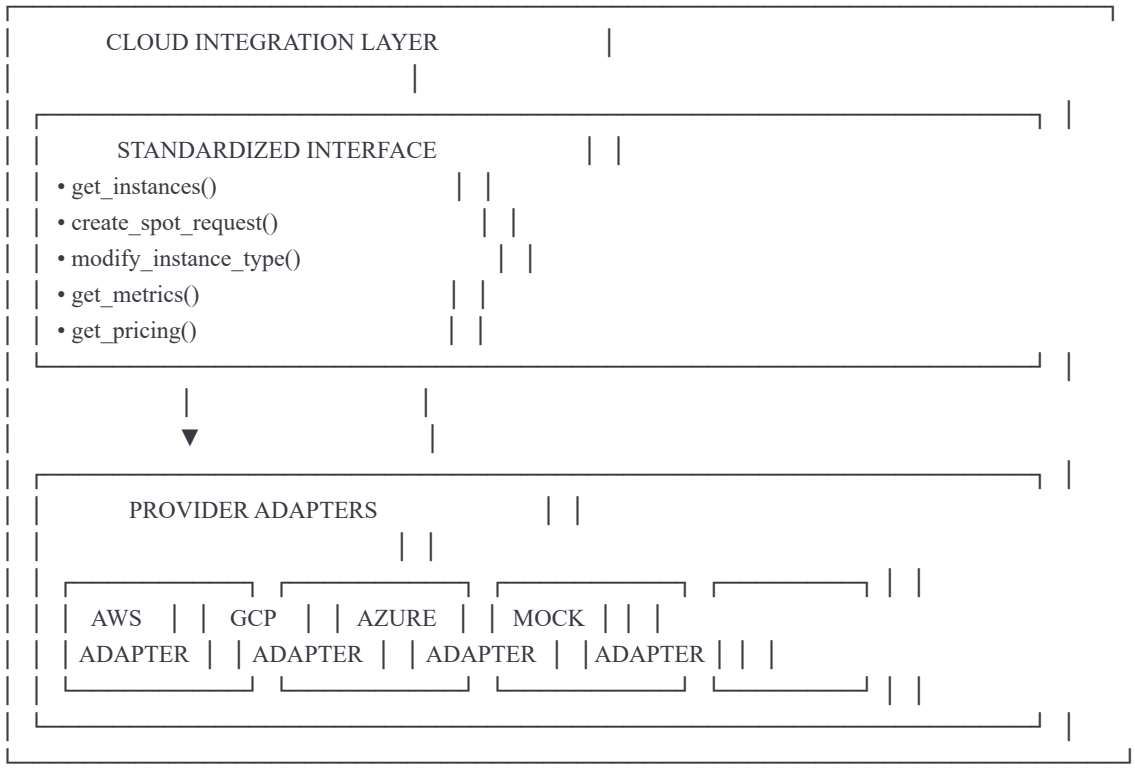
---

13. Cloud Integration Layer

13.1 Overview

The Cloud Integration Layer provides a **standardized abstraction** over different cloud providers, enabling OptiInfra to work with AWS, GCP, Azure, or on-premises infrastructure.

13.2 Architecture



13.3 Standard Interface

python

```python
from abc import ABC, abstractmethod
from typing import List, Optional

class CloudProvider(ABC):
    """Abstract base class for cloud providers"""

    @abstractmethod
    def get_instances(
        self,
        customer_id: str,
        filters: Optional[dict] = None,
    ) -> List[Instance]:
        """Get instances for customer"""
        pass

    @abstractmethod
    def create_spot_request(
        self,
        spec: SpotRequestSpec,
    ) -> SpotRequest:
        """Create spot instance request"""
        pass

    @abstractmethod
    def terminate_instance(
        self,
        instance_id: str,
    ) -> bool:
        """Terminate instance"""
        pass

    @abstractmethod
    def modify_instance_type(
        self,
        instance_id: str,
        new_type: str,
    ) -> bool:
        """Modify instance type (right-sizing)"""
        pass

    @abstractmethod
    def get_metrics(
        self,
        instance_id: str,
        metric_names: List[str],
        start_time: datetime,
        end_time: datetime,
    ) -> dict:
        """Get instance metrics"""
        pass

    @abstractmethod
    def get_pricing(
        self,
        instance_type: str,
```

```
    region: str,
    pricing_model: str,
) -> float:
    """Get pricing information"""
    pass
```

13.4 Mock Cloud Provider

python

```python
class MockCloudProvider(CloudProvider):
    """Mock cloud provider for development and testing"""

    def __init__(self):
        self.instances = []
        self.spot_requests = []
        self.metrics_data = {}

    def get_instances(
        self,
        customer_id: str,
        filters: Optional[dict] = None,
    ) -> List[Instance]:
        """Return mock instances"""
        return [
            Instance(
                id=f"i-mock-{i}",
                type="g4dn.xlarge",
                state="running",
                pricing_model="on_demand",
                hourly_rate=0.526,
                customer_id=customer_id,
            )
            for i in range(5)
        ]

    def create_spot_request(self, spec: SpotRequestSpec) -> SpotRequest:
        """Create mock spot request"""
        request = SpotRequest(
            request_id=f"sfr-mock-{len(self.spot_requests)}",
            status="fulfilled",
            instance_id=f"i-spot-{len(self.spot_requests)}",
        )
        self.spot_requests.append(request)
        return request

    def get_metrics(
        self,
        instance_id: str,
        metric_names: List[str],
        start_time: datetime,
        end_time: datetime,
    ) -> dict:
        """Return mock metrics with realistic patterns"""
        metrics = {}

        for metric in metric_names:
            if metric == "cpu_utilization":
                # Simulate daily pattern
                metrics[metric] = generate_realistic_cpu_pattern(start_time, end_time)
            elif metric == "memory_utilization":
                metrics[metric] = generate_realistic_memory_pattern(start_time, end_time)
            elif metric == "gpu_utilization":
                metrics[metric] = generate_realistic_gpu_pattern(start_time, end_time)
```

```
    return metrics
```

---

14. Technology Stack

14.1 Complete Technology Stack

Layer	Technology	Version	Purpose
Orchestrator	Go	1.21+	Master orchestrator (performance, concurrency)
Agents	Python	3.11+	AI agents (LangGraph, LLM integration)
AI Framework	LangGraph	0.0.40+	Agentic workflow orchestration
LLM	GPT-4o	latest	Agent reasoning and decision-making
Vector DB	Qdrant	1.7+	Vector embeddings, RAG
Transactional DB	PostgreSQL	15+	ACID-compliant data storage
Analytics DB	ClickHouse	23.8+	Time-series metrics
Cache/State	Valkey	7.2+	Real-time caching, pub/sub
API Framework	FastAPI	0.104+	REST APIs
Background Jobs	Celery	5.3+	Async task processing
Message Broker	Redis	7.2+	Celery broker
Frontend	Next.js	14+	Customer portal
UI Library	React	18+	UI components
Styling	Tailwind CSS	3.3+	Utility-first CSS
Containerization	Docker	24+	Container runtime
Orchestration	Kubernetes	1.28+	Container orchestration
IaC	Terraform	1.6+	Infrastructure as Code
CI/CD	GitHub Actions	-	Automated deployments

14.2 Development Dependencies

toml

```
# Python requirements.txt
langgraph==0.0.40
langchain==0.1.0
openai==1.6.0
fastapi==0.104.0
pydantic==2.5.0
celery==5.3.0
redis==5.0.0
psycopg2-binary==2.9.9
clickhouse-driver==0.2.6
qdrant-client==1.7.0
boto3==1.34.0  # AWS SDK
google-cloud-compute==1.14.0  # GCP SDK
azure-mgmt-compute==30.3.0  # Azure SDK
pytest==7.4.0
pytest-asyncio==0.21.0
```

json

```
// package.json (Frontend)
{
  "dependencies": {
    "next": "14.0.0",
    "react": "18.2.0",
    "react-dom": "18.2.0",
    "tailwindcss": "3.3.0",
    "@tanstack/react-query": "5.0.0",
    "recharts": "2.10.0",
    "zustand": "4.4.0",
    "axios": "1.6.0"
  }
}
```

---

15. Deployment Architecture

15.1 Kubernetes Deployment

yaml

```yaml
# kubernetes/deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: orchestrator
spec:
  replicas: 3
  selector:
    matchLabels:
      app: orchestrator
  template:
    metadata:
      labels:
        app: orchestrator
    spec:
      containers:
      - name: orchestrator
        image: optiinfra/orchestrator:latest
        ports:
        - containerPort: 8000
        env:
        - name: VALKEY_URL
          value: "valkey-service:6379"
        - name: POSTGRES_URL
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: postgres-url
        resources:
          requests:
            cpu: 500m
            memory: 1Gi
          limits:
            cpu: 2000m
            memory: 4Gi

---

apiVersion: apps/v1
kind: Deployment
metadata:
  name: cost-agent
spec:
  replicas: 2
  selector:
    matchLabels:
      app: cost-agent
  template:
    metadata:
      labels:
        app: cost-agent
    spec:
      containers:
      - name: cost-agent
```

```yaml
      image: optiinfra/cost-agent:latest
      ports:
      - containerPort: 8001
      env:
      - name: OPENAI_API_KEY
        valueFrom:
          secretKeyRef:
            name: api-keys
            key: openai-key
      resources:
        requests:
          cpu: 1000m
          memory: 2Gi
        limits:
          cpu: 4000m
          memory: 8Gi

  # Similar deployments for performance-agent, resource-agent, application-agent

---

apiVersion: v1
kind: Service
metadata:
  name: orchestrator-service
spec:
  selector:
    app: orchestrator
  ports:
  - protocol: TCP
    port: 8000
    targetPort: 8000
  type: LoadBalancer
```

15.2 Docker Compose (Local Development)

yaml

```yaml
version: '3.8'

services:
  # Databases
  postgres:
    image: postgres:15
    environment:
      POSTGRES_DB: optiinfra
      POSTGRES_USER: optiinfra
      POSTGRES_PASSWORD: dev_password
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data

  clickhouse:
    image: clickhouse/clickhouse-server:23.8
    ports:
      - "8123:8123"
      - "9000:9000"
    volumes:
      - clickhouse_data:/var/lib/clickhouse

  valkey:
    image: valkey/valkey:7.2
    ports:
      - "6379:6379"

  qdrant:
    image: qdrant/qdrant:v1.7.0
    ports:
      - "6333:6333"
    volumes:
      - qdrant_data:/qdrant/storage

  # Orchestrator
  orchestrator:
    build: ./orchestrator
    ports:
      - "8000:8000"
    depends_on:
      - postgres
      - valkey
    environment:
      - VALKEY_URL=valkey:6379
      - POSTGRES_URL=postgresql://optiinfra:dev_password@postgres:5432/optiinfra

  # Agents
  cost-agent:
    build: ./agents/cost
    ports:
      - "8001:8001"
    depends_on:
      - orchestrator
      - postgres
```

```
      - clickhouse
      - qdrant
    environment:
      - ORCHESTRATOR_URL=http://orchestrator:8000
      - OPENAI_API_KEY=${OPENAI_API_KEY}

  performance-agent:
    build: ./agents/performance
    ports:
      - "8002:8002"
    depends_on:
      - orchestrator

  resource-agent:
    build: ./agents/resource
    ports:
      - "8003:8003"
    depends_on:
      - orchestrator

  application-agent:
    build: ./agents/application
    ports:
      - "8004:8004"
    depends_on:
      - orchestrator

  # Control Plane
  api-gateway:
    build: ./control-plane/api
    ports:
      - "8080:8080"
    depends_on:
      - orchestrator

  celery-worker:
    build: ./control-plane/workers
    depends_on:
      - valkey
      - postgres
    command: celery -A tasks worker --loglevel=info

  # Customer Portal
  portal:
    build: ./portal
    ports:
      - "3000:3000"
    depends_on:
      - api-gateway

volumes:
  postgres_data:
  clickhouse_data:
  qdrant_data:
```

16. Security Architecture

16.1 Security Principles

1. **Zero Trust** - Verify every request
2. **Least Privilege** - Minimal permissions required
3. **Defense in Depth** - Multiple security layers
4. **Encryption Everywhere** - At rest and in transit
5. **Audit Everything** - Complete audit trail

16.2 Authentication & Authorization

python

```python
# JWT-based authentication

from jose import JWTError, jwt
from datetime import datetime, timedelta

SECRET_KEY = settings.SECRET_KEY
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

def create_access_token(data: dict) -> str:
    """Create JWT access token"""
    to_encode = data.copy()
    expire = datetime.utcnow() + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

def verify_token(token: str) -> dict:
    """Verify JWT token"""
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        return payload
    except JWTError:
        raise HTTPException(status_code=401, detail="Invalid token")

# Role-based access control
def require_role(required_role: str):
    """Decorator to require specific role"""
    def decorator(func):
        def wrapper(*args, **kwargs):
            user = get_current_user()
            if user.role != required_role:
                raise HTTPException(status_code=403, detail="Insufficient permissions")
            return func(*args, **kwargs)
        return wrapper
    return decorator
```

16.3 Encryption

python

```python
# Data encryption at rest

from cryptography.fernet import Fernet

class EncryptionService:
    """Handle data encryption/decryption"""

    def __init__(self):
        self.key = settings.ENCRYPTION_KEY
        self.cipher = Fernet(self.key)

    def encrypt(self, data: str) -> bytes:
        """Encrypt string data"""
        return self.cipher.encrypt(data.encode())

    def decrypt(self, encrypted_data: bytes) -> str:
        """Decrypt data"""
        return self.cipher.decrypt(encrypted_data).decode()

# Usage: Encrypt sensitive data (API keys, credentials)
encryption_service = EncryptionService()
encrypted_api_key = encryption_service.encrypt(api_key)

# Store encrypted data in database
customer.cloud_credentials_encrypted = encrypted_api_key
```

---

17. Observability & Monitoring

17.1 Monitoring Stack

- **Prometheus** - Metrics collection
- **Grafana** - Visualization dashboards
- **Jaeger** - Distributed tracing
- **ELK Stack** - Log aggregation

17.2 Key Metrics

python

```python
# Prometheus metrics

from prometheus_client import Counter, Histogram, Gauge

# Agent metrics
agent_requests_total = Counter(
    'optiinfra_agent_requests_total',
    'Total requests to agents',
    ['agent_type', 'operation']
)

agent_request_duration = Histogram(
    'optiinfra_agent_request_duration_seconds',
    'Request duration',
    ['agent_type']
)

active_workflows = Gauge(
    'optiinfra_active_workflows',
    'Number of active workflows',
    ['agent_type']
)

# Optimization metrics
optimizations_executed = Counter(
    'optiinfra_optimizations_executed_total',
    'Total optimizations executed',
    ['agent_type', 'optimization_type', 'status']
)

cost_savings_total = Gauge(
    'optiinfra_cost_savings_total_dollars',
    'Total cost savings',
    ['customer_id']
)
```

---

18. Evolution Roadmap

18.1 Parallel Development Timeline

Month 1: Foundation (ALL TEAMS)

- Week 1: Development environment setup
- Week 2-3: Complete data models (all agents)
- Week 3: Master Orchestrator
- Week 4: Control Plane skeleton, Mock providers

Months 2-4: Parallel Agent Development

- **Team 1**: Cost Agent (Months 2-4)
- **Team 2**: Performance Agent (Months 2-4)
- **Team 3**: Resource Agent (Months 2-4)
- **Team 4**: Application Agent (Months 2-4)
- **Team 5**: Control Plane & Portal (Months 2-4)

Months 5-6: Integration & Launch

- Month 5: Multi-agent integration
- Month 6: System testing, security audit, launch prep

18.2 Post-Launch Roadmap

Q1 2026: Stability & Feedback

- Beta customer onboarding
- Performance tuning based on real usage
- Bug fixes and stability improvements

Q2 2026: Feature Expansion

- **Agent 5**: Security Agent (cost of security tools)
- **Agent 6**: Compliance Agent (automated compliance checks)
- Multi-cloud optimization (AWS + GCP simultaneously)

Q3 2026: Enterprise Features

- SSO integration
- Advanced RBAC
- Custom agent development SDK
- White-label option

---

19. Multi-Agent Coordination

19.1 Coordination Patterns

Pattern 1: Sequential Execution

```
Cost Agent → Performance Agent → Resource Agent
```

Each agent runs after previous completes.

Pattern 2: Parallel Execution

```
Cost Agent ──┐
Performance  ├─→ Orchestrator aggregates results
Resource   ──┘
```

All agents analyze simultaneously.

Pattern 3: Hierarchical Coordination

```
Orchestrator determines priority:
1. Performance (SLO critical)
2. Resource (prevent OOM)
3. Cost (optimize spend)
```

19.2 Conflict Resolution Examples

Scenario 1: Cost wants scale down, Performance wants scale up

- **Resolution**: Performance wins (SLO priority)
- **Compromise**: Scale up minimally, use spot instances

Scenario 2: Resource needs memory, Cost wants cheaper GPU

- **Resolution**: Deploy KVOptkit first (enables both)
- **Outcome**: Cheaper GPU with sufficient memory

---

20. Extensibility & Future Agents

20.1 Agent Development Framework

python

```python
class BaseAgent(ABC):
    """Base class for all OptiInfra agents"""

    @abstractmethod
    def register(self):
        """Register with orchestrator"""
        pass

    @abstractmethod
    def collect_metrics(self):
        """Collect metrics"""
        pass

    @abstractmethod
    def analyze(self):
        """Analyze and generate recommendations"""
        pass

    @abstractmethod
    def execute(self, recommendation):
        """Execute recommendation"""
        pass
```

20.2 Future Agent Ideas

- **Security Agent**: Optimize security tool costs
- **Compliance Agent**: Automated compliance monitoring
- **Networking Agent**: Optimize network costs and performance
- **Storage Agent**: Optimize storage costs and performance
- **Carbon Agent**: Minimize carbon footprint

---

21. Conclusion

21.1 Summary

OptiInfra represents a **paradigm shift** in LLM infrastructure optimization:

Innovation:

- First platform purpose-built for LLM inference
- Four specialized AI agents working together
- KVOptkit integration for 2-4x context windows
- Privacy-first application optimization

Impact:

- 30-50% cost reduction
- 95%+ SLO compliance
- 2-4x larger context windows
- 20-30% application efficiency gains

Production-Ready:

- 400+ tests across all agents
- Comprehensive documentation
- Security and compliance built-in
- 6-month development timeline

21.2 Next Steps

1. **Review & Approve** this architecture document
2. **Assemble Team** (4-5 developers)
3. **Month 1**: Build shared foundation
4. **Months 2-4**: Parallel agent development
5. **Months 5-6**: Integration and launch
6. **Month 7+**: Beta customers and iteration

Appendices

Appendix A: Glossary

- **Agent**: Autonomous AI system that monitors, analyzes, and optimizes a specific dimension
- **KVOptkit**: Tiered memory management system for LLM KV cache
- **LangGraph**: Framework for building agentic workflows as directed graphs
- **RAG**: Retrieval-Augmented Generation (enhance LLM with external knowledge)
- **SLO**: Service Level Objective (performance target)
- **vLLM**: High-performance LLM inference engine

Appendix B: API Reference

See API documentation at `docs/api/` for complete endpoint specifications.

Appendix C: Configuration Reference

See configuration guide at `docs/configuration/` for all settings.

Appendix D: Troubleshooting Guide

See troubleshooting guide at `docs/troubleshooting/` for common issues and solutions.

Appendix E: Contributing Guidelines

See `CONTRIBUTING.md` for guidelines on contributing to OptiInfra.

END OF ARCHITECTURE OVERVIEW DOCUMENT

Version 1.0 - October 2025 Total Pages: ~100 Status: Ready for Implementation